ESD-TR-70-198

MTR-1843

PRINCIPLES OF OPERATION OF THE VENUS MICROPROGRAM

B. J. Huberman

JULY 1970

Prepared for

DIRECTORATE OF PLANNING AND TECHNOLOGY
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 700A
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract F19(628)-68-C-0365

ESD-TR-70-198

MTR-1843

PRINCIPLES OF OPERATION OF THE VENUS MICROPROGRAM

B. J. Huberman

JULY 1970

Prepared for

DIRECTORATE OF PLANNING AND TECHNOLOGY
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 700A
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract F19(628)-68-C-0365

FOREWORD

This report was developed under Air Force Contract F19(628)-68-C-0365 with The MITRE Corporation in Bedford, Massachusetts. It carries a MITRE Project Number of 700A. There is no Air Force project or task number. The report describes the Venus system, which is a computer system comprised of microprograms and software. It is implemented on the Interdata 3, a small, micro-programmable computer. This document contains a complete description of the micro-program part of Venus.

REVIEW AND APPROVAL

This technical report has been reviewed and is approved.

Robert P. Savoy

for

ANTHONY P. TRUNFIO
Technical Advisor
Development Engineering Division
Directorate of Planning & Technology

ABSTRACT

Venus is a computer system comprised of microprograms and software. It is implemented on the Interdata 3, which is a small, microprogrammable computer. This document contains a complete description of the microprogram part of Venus.

TABLE OF CONTENTS

TABLE OF CONTENTS (Concluded)

LIST OF ILLUSTRATIONS

SECTION I

INTRODUCTION

Venus is a computer system comprised of microprograms and software. It is implemented on the Interdata 3, which is a small, microprogrammable computer. The Venus system supports:

1. Multiprogramming;
2. Named virtual memories;
3. Recursive and reentrant procedures;
4. Interrupts for debugging;
5. A microprogrammed, multiplex channel.

This document contains a complete description of the microprogram part of Venus. Each of the above features is partially or fully implemented in the microprogram. In addition, the microprogram part of Venus completely defines the instruction set (used by the software part). The software part of Venus is not defined by this document; occasionally, however, a place will be described where the microprogram and the software communicate.

This document is intended to be read sequentially the first time, since the descriptions of important concepts in the Venus system appear as required for the definitions of instructions. Subsequently, the document may be used as a reference manual. Appendix II provides an index to the instructions.

SECTION II

NOTATION

NUMBER BASES

Venus reflects the fact that the Interdata is a hexadecimal machine, addressable by 8-bit bytes. Each byte contains two hexadecimal digits. In this document, hexadecimal numbers will be used for locations and for constants, like FF. Decimal numbers are used to refer to quantities (for example, 16 general registers) and to refer to bits.

ADDRESS SPACES

In Venus there are three address spaces: registers, core and streams. Locations in the various address spaces are represented as follows.

Registers

A user in Venus is supplied with 16, 16-bit general registers. The name of a register is a 4-bit number (hexadecimal digit); e.g., 2 or F. In this document registers are referred to symbolically by the letters X and R.

Associated with each register is a stream register. The stream register is named symbolically by putting an S before the symbolic name of the general register:

SR is the stream register associated with register R.

SX is the stream register associated with register X.

Core

Any 16-bit quantity can stand for a location in core. Core is divided into 256-byte pages. CP is frequently used as an abbreviation for core page.

Streams

Streams are named virtual memories. In order to specify a location in a stream, both the name of the stream (15 bits) and the location within the stream (16 bits) must be given. The following notation is used:

stream name.address in stream .

Streams are divided into 256-byte pages. SP is frequently used as an abbreviation for stream page.

VALUES

Data in Venus occurs in 8- or 16-bit containers. Bytes are 8-bit containers while halfwords are 16 bits long. Bits in a halfword are numbered from 0 to 15. Core and streams are both addressed at byte boundaries. A halfword consists of two bytes and always starts on an even byte boundary:

0 7	8 15
byte n	byte n + 1

, where n is even.

halfword n

If a reference is made to halfword m, where m is odd, then halfword m-1 will be accessed.

No particular meaning is assigned to bytes. However, halfwords contain different kinds of data.

Numeric Quantities

All numeric quantities are 16 bits long. A 16-bit quantity may stand for a signed 15-bit integer, an unsigned 16-bit integer, or an address in core or within a stream. Signed integers are all stored in two's complement notation. Bit 0 is the sign bit.

Stream Names

Stream names are 15 bits long but are contained in a 16-bit halfword with bit zero unused:

```
 0 1              15
|  |  stream name  | .
 ↑
unused
```

Semaphores

Semaphores are 7-bit signed integers. They are always located in the right byte of a 16-bit halfword; the left byte contains a pointer to the queue associated with the semaphore:

```
|    pointer    |  semaphore  | ,
 0             7 8           15
```

SYNTAX

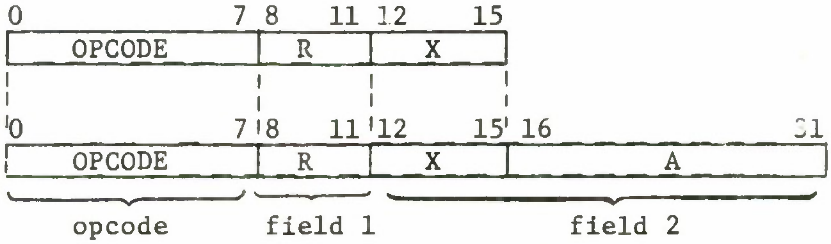
The following terminal characters are used to explain the meaning of instructions:

if	
then	
else	
begin	
end	
rotated left	
+	plus
-	minus
mod	modulo
.	stream address

$*$	times
$\wedge$	and
$\vee$	or
$\neg$	not
<u>xor</u>	exclusive or
$=$	equal
$\neq$	not equal
;	end of statement
$\overline{2E}$	one's complement; for example, $\overline{2E} = D1$
$\leftarrow$	assignment
( )	"the contents of"; for example, (R) means the contents of general register R (A) means the contents of core location A
R_{8-15} R_6	subscripts are used whenever less than 16 bits of a 16-bit quantity are used; e.g., R_{8-15} means bits 8 through 15 of register R, and R_6 means bit 6 of register R
$\lvert\ \rvert$	used to clarify the extent of an operation

NOTATION FOR INSTRUCTIONS

Instructions in Venus consist of an opcode and two fields. They occupy one or two halfwords.



The opcode is 8 bits long. The first field, R, is 4 bits long and usually contains the number of a general register. The second field is either 4 bits long or 20 bits long. If it is 4 bits long, it contains the number, X, of a general register and the instruction is called a short instruction. A short instruction is written:

OPCODE R,X .

If the second field is 20 bits long, it is divided into two subfields, A and X. X is the number of a general register and A is a 16-bit quantity. Such an instruction is called a long instruction and is written:

OPCODE R,A(X).

A and X are combined together to form a 16-bit address or a 16- or 8- or 4-bit value.

Some instructions make use of only one field. The same notation is used for fields in this case, so the reader can tell which field is being used and which is omitted. For example:

JOBA	R	is a short instruction using only the first field.
P	A(X)	is a long instruction using only the second field.

The notation for instructions is introduced here only for the purposes of this document. It is not related to any language running under Venus.

CLASSES OF INSTRUCTION

Most instructions in Venus are members of one of the following five classes. Membership in a class determines what the second field of the instruction means.

Class C: Core instructions. Instructions in this class are all long (32 bits). The second field defines a location in core.

Class R: Register instructions. Instructions in this class are all short (16 bits) and the last letter of the opcode mnemonic is usually R. The second field contains the number of a general register.

Class I: Immediate instructions. These instructions are all long and the last letter of the opcode mnemonic is usually I. The second field defines an 8- or 16-bit constant.

Class S: Stream instructions. These instructions are all long and the last letter of the opcode mnemonic is always S. The second field defines a location in a stream.

Class P: Procedure instructions. These instructions are all long and the last letter of the opcode mnemonic is always P. The second field defines a location within the current procedure stream (the procedure which contains the instruction being executed). The name of this procedure is stored in JSTRNM in the job area.

The user should note that store instructions only occur in classes C, R and S. In class I the second field does not provide a location in which to store things. In class P the second field provides a location in a procedure; however, Venus encourages the writing of reentrant procedures by not providing instructions to store in procedures.

CONDITION CODE

The condition code contains information about the effect of a previous instruction. Many instructions set it; some instructions test it. It is composed of four bits: the C, V, G and L bits.

condition code=	C	V	G	L

In general these bits have the following meaning:

- C is the "carry" bit. In general it is on if a carry occurred (a 1 was carried out of bit 0) during the execution of a previous instruction.

- V is the "arithmetic overflow" bit. If it is on, then during the execution of some previous instruction the bit carried out of position 1 was not the same as the bit carried out of position 0.

- G is the "greater than zero" bit. If it is on, it means the result of the execution of some previous instruction had bit 0 (the sign bit) off.

- L is the "less than zero" bit. If it is on, then the result of the execution of some previous instruction had bit 0 (the sign bit) on.

SECTION III

THE JOB AREA

Venus is a multiprogramming system in which 16 jobs can run concurrently. These jobs are numbered in hex from 0 to F. A job area consisting of 156 contiguous bytes of core memory is assigned to each job running under the VENUS system. The first location in the job area for job n is the first location of core page n*10. The core address of this location is n*1000. The job area contains the general registers, instruction counter and all other job specific information. In addition it contains data which is used by the system to make the multiprogramming run smoothly.

Throughout the body of this paper, references are made to data in the job area. These references use the symbolic names whose locations are defined in the chart in Appendix III. The true location of any piece of data, D, for job n is

$$\left\{ n*1000 \right\} + D.$$

For example, the location of the LINK register for job 3 is

$$\left\{ n*1000 \right\} + LINK = \left\{ 3*1000 \right\} + 6A = 306A.$$

THE JOB AREA INSTRUCTION

Most instructions which refer to data in the job area are automatically interpreted by the microprogram to refer to the job area of the job executing the instruction. Sometimes, however, it is necessary for a job to know the location of the job area from which it is running. The JOBA instruction provides this information.

<u>Job Area Instruction</u>

```
JOBA     R
```

As the result of the execution of JOBA, the core address of the first byte of the job area is stored in register R. For example, if job A performs

```
JOBA     R
```

then the contents of register R become A000.

The condition code is not affected by the JOBA instruction.

SECTION IV

VIRTUAL MEMORY

Most data used by a program in Venus will be stored in streams. Streams are named virtual memories, containing 64K bytes of data and having 15-bit names. Streams are divided into 256-byte pages. Core is also divided into 256-byte pages, so that one stream page fits into one core page. The Interdata is supplied with a disk. Streams are paged between disk and core and the microprogram automatically maps stream addresses into core addresses.

When accessing data within a stream, it is necessary to give the name of the stream and the 16-bit address within the stream. In order to provide space for this data, each job is provided with eight 16-bit stream registers. The stream registers are paired with the general registers: a stream register, general register pair provides the bits needed to reference a stream. The stream register holds the stream name, right adjusted, while the general register holds the address within the stream. Given the number of the general register, X, the number of the associated stream register, SX, is computed simply:

$$SX = X \underline{\text{mod}}\ 8.$$

Each stream register is equipped with a 16-bit extension which is also located in the job area. This extension is used by the microprogram to hold information about the mapping of stream addresses into core addresses. Figure 1 shows the relationship between registers, stream registers and extensions.

MAPPING OF VIRTUAL INTO REAL MEMORY

When a reference is made to a stream, only the general register, X, is mentioned. The microprogram computes the number of the associated stream register, SX, and uses its contents as the stream name, SN. This means that prior to making a reference to a stream, a program must load the stream register being used for the reference with the name of the stream being referenced. The value of the address within the stream, SA, depends upon the instruction being executed.

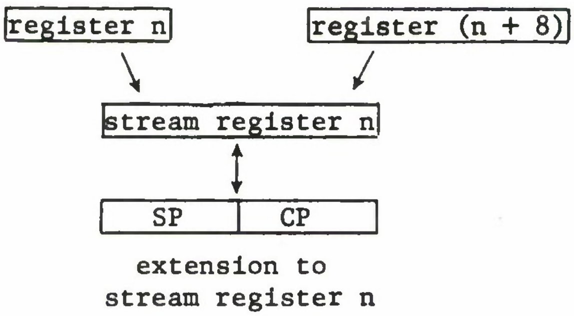


Figure 1. The Relationship between Registers, Stream Registers and Extensions

SA is broken into two parts:

SA =

0 — 7	8 — 15
SP_A	LOW

,

where SP_A is the stream page and LOW is the address within the stream page. The extension of SX is also broken into two parts:

extension =

0 — 7	8 — 15
SP	CP

,

where SP is a stream page number and CP is a core page number. If $CP \neq 0$, then stream page SP of the stream whose name is in SX is contained in core page CP. Therefore, if

$$CP \neq 0 \wedge SP_A = SP, \qquad (1)$$

the core address of the desired data is simply

core address =

CP	LOW

.

The desired stream page in this case is called a locked-in page because it is guaranteed to be located in core at core page CP. The reason that this particular page is locked in is because previously a stream reference was made to it through stream register SX. This fact is central to the way programs run in Venus.

If a stream page is in core, the microprogram will find it. However, the amount of time required to locate it varies. If it is locked-in by the stream register being used to reference it, the minimum amount of time is required. Otherwise, the microprogram must search for it. The remainder of this section describes the details of this search.

Whenever (1) above is not true, the microprogram makes use of a table in core called the Core Page Table (CPT). This table contains information about the contents of each core page. Among other things, it contains the stream name and the number of the stream page which occupy a core page. The CPT is indexed by core page and entered by means of a hash chain from hash table HASH. An 8-bit hash code, H, is formed:

$$H \leftarrow \left| SN \wedge FF \right| \underline{\text{xor}}\ SP_A.$$

H is used as an entry into HASH and a search is made of the hash chain starting from $HASH_H$ to see if the desired stream page is contained in core. If it is in core, say at core page CPI, then the core address is

core address = $\boxed{\quad \text{CP1} \quad}\boxed{\quad \text{LOW} \quad}$.

In addition to forming the core address, the microprogram updates the contents of the extension of stream register SX to contain information about this reference (it locks in the stream page). It becomes

extension = $\boxed{\quad SP_A \quad}\boxed{\quad \text{CPI} \quad}$.

It is likely, of course, that the desired stream page is not in core. This is recognized when the microprogram reaches the end of the hash chain without finding the desired page. The end of the hash chain is recognized by finding a link equal to 0. In this case the microprogram stops right where it is, in the middle of executing an instruction, and starts a software program called the page fault handler (PFH). The PFH will fetch the desired page from the disk, store it in some core page, and update the Core Page Table and the HASH table. Then it returns, via a special instruction (UNQP), to the microprogram at the place from which it was started.

THE AGE CHAIN

In order to bring a page into core, the PFH will very likely have to remove a page from core. Of course, it cannot remove any page which is locked in. Instead it must remove some page which is not referred to by any stream register extension. It does not need to search the Core Page Table for these pages, however, because the microprogram places all pages which are not locked in on an age chain. The head of the age chain is AGEHD at location 202:

AGEHD = $\boxed{\quad \text{OLDEST} \quad}\boxed{\quad \text{NEWEST} \quad}$.

OLDEST is the number of the core page containing the stream page which was released from being locked in longest ago, while NEWEST is the number of the core page containing the stream page which was most recently released.

The microprogram is responsible for keeping all pages which are not locked in on the age chain. Two functions are performed by the microprogram to keep the age chain current: *aged* and *unaged*. *aged* is performed whenever a reference is made through a stream register whose extension locks in a different page than the desired one, while *unaged* is performed when the search through the Core Page Table via the hash chain successfully locates the desired stream page.

The definitions of *aged* and *unaged* take account of the fact that in Venus streams are shared. This is obvious from the way the microprogram searches the CPT for a desired stream page. Any job which wants to use a stream simply refers to it by name. This means that when a reference is made through a stream register whose extension locks in a different page, the microprogram cannot automatically put that page on the age chain because it may be referred to by some other stream register extension. Therefore, the microprogram keeps a count for each locked-in page of how many different extensions lock it in.

THE CORE PAGE TABLE

Before definitions of *aged* and *unaged* are given, the Core Page Table must be defined. The CPT is located in core starting at location 200 and extending to 7FF. The table is actually divided into three separate tables. Each of these tables is indexed by core page. The user should note, however, that not all pages in core are available to hold stream pages. Among others, pages are reserved for the job areas and for data used by the microprogram, including the HASH table and the CPT. Spaces thus appear in the CPT and these are sometimes used to hold other information. This means that the CPT can be searched meaningfully only through the hash chains.

For core pages which contain stream pages, the CPT contains the following information.

1. SN table. This table is located between 400 and 5FF. For each core page, there is one halfword of information:

0	1 15
	SN

↑
Change Bit

Bit 0 of this halfword is set by the microprogram whenever data is stored in the core page. SN is the name of the stream whose page occupies this core page.

2. SPNEXT table. This table is located between 600 and 7FF. For each core page, there is one halfword of information:

0 — 7	8 — 15
SP	NEXT

SP is the number of the stream page which occupies this core page. NEXT is the core page of the next link on the hash chain. If NEXT = 0, this entry is the end of the chain.

3. INDCNT table. This table is located between 200 and 3FF. For each core page, there is one halfword of information:

0 — 7	8 — 15
IND	CNT

This information has two different meanings, depending on whether the stream page is locked in or not.

a. Locked in. The page is locked in if

$$IND = 0.$$

In this case CNT equals one less than the number of extensions locking in the page.

b. Aged. The page is on the age chain if

$$IND \neq 0.$$

In this case IND equals the core page of the next newer page on the age chain and CNT equals the number of the next older page.

Figure 2 gives an example of how HASH and CPT are used.

HASH Table

	⋮
entry 21	0 0
22	3 8
23	3 6
	⋮

Core Page Table

core page	IND	CNT	SN	SP	NEXT
			⋮		
36	0 0	0 0	0 1 D C	F F	3 7
37	0 1	3 8	4 1 2 3	0	0
38	3 7	0 1	0 1 D C	F E	0
			⋮		

1. Three entries in HASH are shown. $HASH_{21} = 0$, which means no stream page located in core at the moment hashes into 21. $HASH_{22}$ and $HASH_{23}$ both contain the starts of hash chains. The chain coming from $HASH_{22}$ is only one link long ($HASH_{22} = 38$ and $NEXT_{38} = 0$). The chain coming from $HASH_{23}$ is two links long ($HASH_{23} = 36$, $NEXT_{36} = 37$, and $NEXT_{37} = 0$).

2. Core page 36 contains stream 1DC, page FF, and core page 37 contains stream 4123, page 0. Both these stream pages produce the same hash code. Core page 38 contains stream 1DC, page FE. Since $HASH_{21} = 0$, stream 1DC, page FD is not in core.

3. Stream 1DC, page FF is locked in ($IND_{36} = 0$) and has one user ($CNT_{36} = 0$). The other two pages are on the age chain. Stream 1DC, page FE has been on the chain longer than stream 4123, page 0.

Figure 2. Example of Entries in Core Page Table

Definitions of aged and unaged

Now definitions can be given for aged and unaged. If a stream reference is made through register X and

$$CP \neq 0 \wedge SP \neq SP_A,$$

then the stream page location in core page CP is aged. This means:

$$(CNT_{CP}) \leftarrow (CNT_{CP}) - 1;$$

$$\underline{\text{if}}\ (CNT_{CP}) < 0\ \underline{\text{then}}\ \underline{\text{do}}\ \underline{\text{begin}}$$

$$(CNT_{CP}) \leftarrow (NEWEST);$$

$$(IND_{CP}) \leftarrow 1;$$

$$(IND_{(NEWEST)}) \leftarrow CP;$$

$$(NEWEST) \leftarrow CP\ \ \underline{\text{end}}$$

When the desired stream page is found in core at core page CP1, it is unaged. This means:

$$\underline{\text{if}}\ (IND_{CP1}) = 0\ \underline{\text{then}}\ (CNT_{CP1}) \leftarrow (CNT_{CP1}) + 1\ \underline{\text{else}}$$

$$\underline{\text{do}}\ \underline{\text{begin}}$$

$$(CNT_{(IND_{CP1})}) \leftarrow (CNT_{CP1});$$

$$(IND_{(CNT_{CP1})}) \leftarrow (IND_{CP1});$$

$$(IND_{CP1}) \leftarrow 0;$$

$$(CNT_{CP1}) \leftarrow 0\ \ \underline{\text{end}}$$

STREAM REGISTER INSTRUCTIONS

When a reference is made to a stream, only the general register, X, is mentioned. The microprogram computes the name of the associated stream register, SX, and uses its contents as the stream name. This means that prior to making a reference to a stream, a program must load the stream register, SX, being used for the reference with the name of the stream being referenced. The "load stream name" instructions are defined for this purpose. The "store stream name" instructions are used to save the contents of stream registers.

<u>Load Stream Name</u>

The second field defines a stream name. This name is compared with the current contents of the stream register, SR, associated with the general register R. If the names are the same, the instruction is finished. If the names are different and some stream page is locked in by the extension of SR, that stream page is <u>aged</u>. The core page in the extension is set to zero. The stream name is then loaded into SR. The condition code is not affected.

LSN R,A(X) Load stream name from core

$(SR) \leftarrow \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X)) \right|$

LSNR R,X Load stream name from register

$(SR) \leftarrow (X)$

LSNI R,A(X) Load stream name immediate

$(SR) \leftarrow \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$

LSNS R,A(X) Load stream name from stream

$(SR) \leftarrow ((SX).\left| A + (X) \right|)$

LSNP R,A(X) Load stream name from procedure

$(SR) \leftarrow ((JSTRNM).\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|)$

Store Stream Name

The contents of the stream register, SR, associated with R are stored in the location specified by the second field. The condition code is not affected.

SSN R,A(X) Store stream name in core

<u>if</u> X = 0 <u>then</u> (A) ← (SR) <u>else</u> (A + (X)) ← (SR)

SSNR R,X Store stream name in register

(X) ← (SR)

SSNS R,A(X) Store stream name in stream

$((SX).\left| A + (X) \right|) \leftarrow (SR)$

SECTION V

BASIC INSTRUCTIONS

LOAD AND STORE INSTRUCTIONS

These instructions allow the user to move data from one location to another.

Load Halfword

The value of the second field defines a 16-bit halfword. It is loaded into the general register R. Load instructions set the condition code as follows:

C	V	G	L	
0	0	0	1	$(R_0) = 1$
0	0	1	0	$(R_0) = 0 \wedge (R) \neq 0$
0	0	0	0	$(R) = 0$

LH R,A(X) Load halfword from core

$$(R) \leftarrow \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X)) \right|$$

LHR R,X Load halfword from register

$$(R) \leftarrow (X)$$

LHI R,A(X) Load halfword immediate

$$(R) \leftarrow \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$$

LHS R,A(X) Load halfword from stream

$$(R) \leftarrow ((SX).\left| A + (X) \right|)$$

LHP R,A(X) Load halfword from procedure

$$(R) \leftarrow ((JSTRNM).\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|)$$

Load Byte

The value of the second field defines an 8-bit byte. It is stored in the right byte of R. The left byte of R is set to 0. The condition code is not affected.

LB R,A(X) Load byte from core

$(R_{0-7}) \leftarrow 0$

$(R_{8-15}) \leftarrow \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X)) \right|$

LBR R,X Load byte from register

$(R_{0-7}) \leftarrow 0$

$(R_{8-15}) \leftarrow (X_{8-15})$

LBI R,A(X) Load byte immediate

$(R_{0-7}) \leftarrow 0$

$(R_{8-15}) \leftarrow \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A_{8-15}\ \underline{\text{else}}\ \left| A + (X) \right|_{8-15} \right|$

LBS R,A(X) Load byte from stream

$(R_{0-7}) \leftarrow 0$

$(R_{8-15}) \leftarrow ((SX).\left| A + (X) \right|)$

LBP R,A(X) Load byte from procedure

$(R_{0-7}) \leftarrow 0$

$(R_{8-15}) \leftarrow ((JSTRNM).\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|)$

Store Halfword

The contents of R are stored in the location indicated by the second field. The condition code is not affected.

STH R,A(X) Store halfword in core

if $X = 0$ *then* $(A) \leftarrow (R)$ *else* $(A + (X)) \leftarrow (R)$

STHR R,X Store halfword in register

$(X) \leftarrow (R)$

STHS R,A(X) Store halfword in stream

$((SX).|A + (X)|) \leftarrow (R)$

Store Byte

The contents of R_{8-15} are stored in the location specified by the second field. The condition code is not affected.

STB R,A(X) Store byte in core

if $X = 0$ *then* $(A) \leftarrow (R_{8-15})$ *else* $(A + (X)) \leftarrow (R_{8-15})$

STBR R,X Store byte in register

$(X_{8-15}) \leftarrow (R_{8-15})$

STBS R,A(X) Store byte in stream

$((SX).|A + (X)|) \leftarrow (R_{8-15})$

ARITHMETIC INSTRUCTIONS

Addition and subtraction are the only arithmetic instructions supported by the Venus microprogram. These instructions operate on halfwords of data.

Add Halfword

The second field defines a 16-bit value. It is added to the contents of R. The result is stored in R. The condition code becomes:

C	V	G	L	
		1	0	$(R_0) = 0 \wedge (R) \neq 0$
		0	1	$(R_0) = 1$
		0	0	$(R) = 0$
	1			Arithmetic overflow
1				Carry

AH R,A(X) Add halfword from core

$$(R) \leftarrow (R) + \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X)) \right|$$

AHR R,X Add halfword from register

$$(R) \leftarrow (R) + (X)$$

AHI R,A(X) Add halfword immediate

$$(R) \leftarrow (R) + \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$$

AHS R,A(X) Add halfword from stream

$$(R) \leftarrow (R) + ((SX).\left| A + (X) \right|)$$

AHP R,A(X) Add halfword from procedure

$$(R) \leftarrow (R) + ((JSTRNM).\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|)$$

Subtract Halfword

The second field defines a 16-bit value. It is subtracted from the contents of R. The result is stored in R. The condition code becomes:

C	V	G	L	
		1	0	$(R_0) = 0 \wedge (R) \neq 0$
		0	1	$(R_0) = 1$
		0	0	$(R) = 0$
	1			Arithmetic overflow
1				Borrow

SH R,A(X) Subtract halfword from core

$(R) \leftarrow (R) - \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X)) \right|$

SHR R,X Subtract halfword from register

$(R) \leftarrow (R) - (X)$

SHI R,A(X) Subtract halfword immediate

$(R) \leftarrow (R) - \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$

SHS R,A(X) Subtract halfword from stream

$(R) \leftarrow (R) - ((SX).\left| A + (X) \right|)$

SHP R,A(X) Subtract halfword from procedure

$(R) \leftarrow (R) - ((JSTRNM).\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|)$

Programmer's Note: The result of the subtraction is in two's complement form. For example, the result of subtracting 3 from 2 is FFFF, and C is set.

Add with Carry Halfword

The second field defines a 16-bit value. This value and the carry bit, C, are added to the contents of R. The result is stored in R. The value of the condition code depends on its value prior to the execution of the instruction:

C	V	G	L	
		1	0	$(R_0) = 0 \wedge \lvert (R) \neq 0 \vee$ the G or L bit was on $\rvert$
		0	1	$(R_0) = 1$
		0	0	$(R_0) = 0 \wedge \lvert$ the G and L bits were off $\rvert$
	1			Arithmetic overflow
1				Carry

ACH R,A(X) Add with carry halfword from core

$(R) \leftarrow (R) + \lvert \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X)) \rvert + C$

ACHR R,X Add with carry halfword from register

$(R) \leftarrow (R) + (X) + C$

ACHI R,A(X) Add with carry halfword immediate

$(R) \leftarrow (R) + \lvert \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \rvert + C$

ACHS R,A(X) Add with carry halfword from stream

$(R) \leftarrow (R) + ((SX).\lvert A + (X) \rvert) + C$

ACHP R,A(X) Add with carry halfword from procedure

$(R) \leftarrow (R) + ((JSTRNM).\lvert \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \rvert) + C$

Programmer's Note: Multiple precision addition may be accomplished by using an "add" instruction for the least significant portion of the two operands and then using an "add with carry halfword" instruction for the remainder of the operation. The result will be in two's complement form. The G and L bits will tell the sign of the multiple precision result.

Subtract with Carry Halfword

The second field defines a 16-bit value. This value and the carry bit, C, are subtracted from the contents of R. The result is stored in R. The value of the condition code depends on its value prior to the execution of the instruction:

C	V	G	L	
		1	0	$(R_0) = 0 \wedge \lvert (R) \neq 0 \vee$ the G or L bit was on $\rvert$
		0	1	$(R_0) = 1$
		0	0	$(R) = 0 \wedge \lvert$ the G and L bits were off $\rvert$
	1			Arithmetic overflow
1				Borrow

SCH R,A(X) Subtract with carry halfword from core

$(R) \leftarrow \lvert (R) - \lvert \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X)) \rvert \rvert - C$

SCHR R,X Subtract with carry halfword from register

$(R) \leftarrow \lvert (R) - (X) \rvert - C$

SCHI R,A(X) Subtract with carry halfword immediate

$(R) \leftarrow \lvert (R) - \lvert \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \rvert \rvert - C$

SCHS R,A(X) Subtract with carry halfword from stream

$(R) \leftarrow \lvert (R) - ((SX).\lvert A + (X) \rvert) \rvert - C$

SCHP R,A(X) Subtract with carry halfword from procedure

$(R) \leftarrow \lvert (R) - ((JSTRNM).\lvert \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \rvert) \rvert - C$

Programmer's Note: See the "add with carry halfword" instructions.

LOGICAL INSTRUCTIONS

The logical instructions permit the user to perform logical "and", "or", and "exclusive or" operations. These operations are always performed on 16-bit unsigned integers. A logical comparison is also available.

And Halfword

A logical "and" is performed on the 16-bit value defined by the second field and the 16-bit contents of R. The result is stored in R. The condition code is set:

C	V	G	L	
0	0	1	0	$(R_0) = 0 \wedge (R) \neq 0$
0	0	0	1	$(R_0) = 1$
0	0	0	0	$(R) = 0$

NH R,A(X) And halfword from core

$(R) \leftarrow (R) \wedge \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X)) \right|$

NHR R,X And halfword from register

$(R) \leftarrow (R) \wedge (X)$

NHI R,A(X) And halfword immediate

$(R) \leftarrow (R) \wedge \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$

NHS R,A(X) And halfword from stream

$(R) \leftarrow (R) \wedge ((SX).\left| A + (X) \right|)$

NHP R,A(X) And halfword from procedure

$(R) \leftarrow (R) \wedge ((JSTRNM).\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|)$

Or Halfword

A logical "or" is performed on the 16-bit value defined by the second field and the 16-bit contents of R. The result is stored in R. The condition code is set:

C	V	G	L	
0	0	1	0	$(R_0) = 0 \wedge (R) \neq 0$
0	0	0	1	$(R_0) = 1$
0	0	0	0	$(R) = 0$

OH R,A(X) Or halfword from core

$$(R) \leftarrow (R) \vee \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X)) \right|$$

OHR R,X Or halfword from register

$$(R) \leftarrow (R) \vee (X)$$

OHI R,A(X) Or halfword immediate

$$(R) \leftarrow (R) \vee \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$$

OHS R,A(X) Or halfword from stream

$$(R) \leftarrow (R) \vee ((SX).\left| A + (X) \right|)$$

OHP R,A(X) Or halfword from procedure

$$(R) \leftarrow (R) \vee ((JSTRNM).\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|)$$

Exclusive Or Halfword

A logical "exclusive or" is performed on the 16-bit value defined by the second field and the 16-bit contents of R. The result is stored in R. The condition code is set:

C	V	G	L	
0	0	1	0	$(R_0) = 0 \wedge (R) \neq 0$
0	0	0	1	$(R_0) = 1$
0	0	0	0	$(R) = 0$

XH R,A(X) Exclusive or halfword from core

$(R) \leftarrow (R)\ \underline{\text{xor}}\ \left|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X))\right|$

XHR R,X Exclusive or halfword from register

$(R) \leftarrow (R)\ \underline{\text{xor}}\ (X)$

XHI R,A(X) Exclusive or halfword immediate

$(R) \leftarrow (R)\ \underline{\text{xor}}\left|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X)\right|$

XHS R,A(X) Exclusive or halfword from stream

$(R) \leftarrow (R)\ \underline{\text{xor}}\ ((SX).\left|A + (X)\right|)$

XHP R,A(X) Exclusive or halfword from procedure

$(R) \leftarrow (R)\ \underline{\text{xor}}\ ((JSTRNM).\left|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X)\right|)$

Compare Logical Halfword

The 16-bit value defined by the second field is compared logically with the contents of R. The different versions of the instruction differ only in the computation of the value. The condition code is set by subtracting the value defined by the second field from (R). This is exactly the same as if the instruction were a subtract. However, only the condition code is affected by the instruction.

CLH R,A(X) Compare logical halfword from core

$$\text{value} = \left|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X))\right|$$

CLHR R,X Compare logical halfword from register

$$\text{value} = (X)$$

CLHI R,A(X) Compare logical halfword immediate

$$\text{value} = \left|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X)\right|$$

CLHS R,A(X) Compare logical halfword from stream

$$\text{value} = ((SX).\left|A + (X)\right|)$$

CLHP R,A(X) Compare logical halfword from procedure

$$\text{value} = ((JSTRNM).\left|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X)\right|)$$

SHIFT INSTRUCTIONS

Shift instructions permit the user to shift a 16-bit halfword to the left or right, or to rotate a 16-bit halfword to the left. In addition, a signed 15-bit integer may be shifted arithmetically to the right.

Rotate Left Halfword

RLH R,A(X)

The contents of R are rotated left the number of bits specified by the second field mod 16. The result is stored in R:

$$(R) \leftarrow (R)\ \underline{\text{rotated left}}\ \Big|\ \big|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X)\big|\ \underline{\text{mod}}\ 16\Big|.$$

The condition code is set as follows:

C	V	G	L	
	0	1	0	$(R_0) = 0 \wedge (R) \neq 0$
	0	0	1	$(R_0) = 1$
0	0	0	0	$(R) = 0$
1	0			$(R_{15}) = 1$
0	0			$(R_{15}) = 0$

Shift Left Halfword Logical

SLHL R,A(X)

The contents of R are shifted left the number of bits specified by the second field mod 16:

$$\text{Amount of shift} = \big|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X)\big|\ \underline{\text{mod}}\ 16.$$

Bits shifted out of position 0 are shifted through the carry bit of the condition code. Zeros are shifted into position 15. The result is stored in R. The condition code is set as follows:

C	V	G	L	
	0	1	0	$(R_0) = 0 \wedge (R) \neq 0$
	0	0	1	$(R_0) = 1$
	0	0	0	$(R) = 0$
1	0			last bit shifted out of position 0 was a 1
0	0			last bit shifted out of position 0 was a 0

Shift Right Halfword Logical

SRHL R,A(X)

The contents of R are shifted right the number of bits specified by the second field mod 16:

$$\text{Amount of shift} = \left|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X)\right| \underline{\text{mod}}\ 16\ .$$

Bits shifted out of position 15 are shifted through the carry bit of the condition code. Zeros are shifted into position 0. The result is stored in R.

The condition code is set as follows:

C	V	G	L	
	0	1	0	$(R_0) = 0 \wedge (R) \neq 0$
	0	0	1	$(R_0) = 1$
	0	0	0	$(R) = 0$
1	0			last bit shifted out of position 15 was a 1
0	0			last bit shifted out of position 15 was a 0

Shift Right Halfword Arithmetic

SRHA R,A(X)

$(R)_{1-15}$ are shifted right the number of bits specified by the second field mod 16:

$$\text{Amount of shift} = \left|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X)\right| \underline{\text{mod}}\ 16\ .$$

The sign bit (R_0) is unchanged. Bits shifted out of position 15 are shifted through the carry bit. The sign bit (R_0) is propagated into position 1, for each bit shifted.

The condition code is set as follows:

C	V	G	L	
	0	1	0	$(R_0) = 0 \wedge (R) \neq 0$
	0	0	1	$(R_0) = 1$
	0	0	0	$(R) = 0$
1	0			last bit shifted out of position 15 was a 1
0	0			last bit shifted out of position 15 was a 0

SECTION VI

CONDITIONS

Conditions represent a set of events which, when enabled by a mask, can cause interrupts in a running job. These interrupts are intended to aid the user in debugging his program. The microprogram or software recognizes the occurrence of an event. The interrupt which results causes the execution of an instruction out of sequence. This instruction can start a software debugging routine. Conditions should not be confused with the condition code.

Conditions are controlled by a 16-bit condition register, COND, and an 8-entry instruction table, with room for one 32-bit instruction per condition. COND is broken into two bytes: ON and MASK.

COND =	ON	MASK

Each bit in ON, if set to 1, indicates the occurrence of a condition. The corresponding bits in MASK are used as a mask. If the bit for an interrupt is set in ON, then the condition is said to be on; if the bit is set in MASK, the condition is enabled.

The microprogram checks for the occurrence of conditions at the beginning of each instruction cycle. If a condition is both on and enabled, an interrupt occurs: the corresponding condition instruction (16 or 32 bits long) in the instruction table is executed, and the condition is turned off. The core location of the instruction which would have been executed if the interrupt had not occurred is stored in TEMPIC. The event which caused the condition bit to be turned on must have happened before the start of the current instruction cycle.

Each condition is assigned a priority, which is reflected in the assignment of the bits, with the highest priority conditions on the left, decreasing toward the right. If more than one on/mask combination is on, the higher priority condition is honored first. If the instruction for this condition is a CALL, as will very frequently be the case, the condition which is being handled and all lower priority conditions (with the exception of the "in-stream illegal instruction" condition) are disabled. They are re-enabled by the execution of the corresponding RETN.

The eight condition bits are assigned as follows:

Bit	Interrupt Condition
0	kill
1	every instruction
2	undefined
3	stack overflow/underflow
4	call
5	undefined
6	illegal in-core instruction
7	illegal in-stream instruction

Kill would normally be turned on by system software when it determines that this job should be terminated. The "kill" condition is always enabled for user programs.

Every instruction is turned on after the execution of every instruction, except the instruction executed as a result of the "every instruction" interrupt. It is turned off by RETN if the return is from a procedure whose execution began as the result of the "every instruction" interrupt.

Undefined occurs only when turned on by software.

Stack overflow/underflow is set when a push or a pop of a stack would store or retrieve from beyond the stack.

Call is set when a CALL instruction is executed, except when the CALL is the instruction to be executed on a "call" condition.

Illegal in-core instruction will be set when an unimplemented instruction is encountered while processing instructions in-core (as opposed to in-stream). The job executing this instruction will normally be running a level 1 program. The microprogram enables this interrupt as well as turning it on.

Illegal in-stream instruction occurs when an unimplemented instruction is encountered in a program running from a procedure stream. This condition is always enabled for programs running from streams, and is used by the microprogram to distinguish between in-core and in-stream programs.

When an illegal instruction is encountered, the instruction counter is stepped by the assumed length of the illegal instruction. The microprogram stores information about the amount of change to the instruction counter in TUTFF in the job area. If bit 6 of TUTFF is on, the program counter has been stepped by 4; otherwise it has been stepped by 2. If the illegal instruction is a condition instruction, the address of this condition instruction relative to the job area is stored in OLDTP. A value for the second field may have been developed before the instruction was discovered to be illegal. This value is stored in TR5R6. A list of the opcodes for which this value is developed is given in Appendix V.

CONDITION INSTRUCTIONS

Two instructions have been defined to modify COND. SET will turn bits on in COND and RSET will turn them off. However, RSET will not turn off either the "kill" bit or the "in-stream" bit. Neither instruction affects the condition code.

Reset Condition Register

RSET R,A(X)

RSET turns off bits in the condition register of the job specified in register R, or the job in which the RSET occurred if R = 0. The bits to be turned off are those which correspond to the bits set in the 16-bit value defined by the second field:

$$(\text{COND}) \leftarrow (\text{COND}) \wedge \left| 81 \vee \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right| \right| .$$

Programmer's Note: The "kill" condition and the "in-stream" condition cannot be disabled.

Set Condition Register

SET R,A(X)

SET turns on bits in the condition register of the job specified in register R, or the job in which the SET occurred if R = 0. The bits turned on correspond to the bits which are on in the 16-bit value defined by the second field:

$$(\text{COND}) \leftarrow (\text{COND}) \vee \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right| .$$

Programmer's Note: When a condition is enabled, an interrupt will take place if the condition occurred in the past. To avoid this, RSET may be used prior to SET to turn off the condition.

SECTION VII

PUSHDOWN STACKS

Streams may be used as pushdown stacks in Venus. When used as a stack, a stream is considered to be a collection of halfwords. Associated with each stack is a stack pointer. A stack pointer is a 32-bit entity contained in a general register, R, and the associated stream register, SR. This pointer always points to the piece of data in the top of the stack.

Stack operations do not affect the condition code. They may, however, turn on the "stack underflow/overflow" bit in the condition register COND. If a stack operation would cause a stack to underflow or overflow, the "underflow/overflow" bit in COND is turned on and the instruction is not executed. In addition, the number of the stream register being used to reference the stack is stored in $SAVREG_{3-5}$ in the job area. That is:

$\underline{\text{if}}$ $\left|(R) = \text{FFFE} \wedge \text{instruction is push}\right| \vee \left|(R) = 0 \wedge \text{instruction is pop}\right|$ $\underline{\text{then}}$

1) The underflow/overflow bit in COND is turned on.

2) The instruction is not executed.

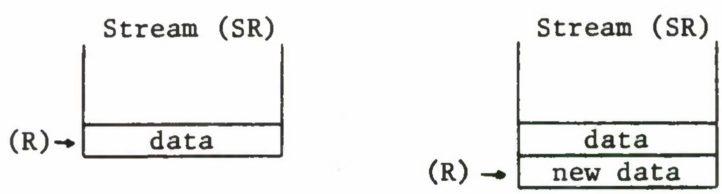
3) $(SAVREG_{3-5}) \leftarrow SR$.

In the following descriptions of the stack instructions, it is assumed that overflow/underflow does not occur.

STACK INSTRUCTIONS

Push Halfword

The contents of R are incremented by two. The 16-bit value defined by the second field is pushed into the stack named by SR at the location specified by the incremented value of R.



before push after push

PU   R,A(X)   Push halfword from core

$(R) \leftarrow (R) + 2$

$((SR).(R)) \leftarrow \left|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A)\ \underline{\text{else}}\ (A + (X))\right|$

PUR   R,X   Push halfword from register

$(R) \leftarrow (R) + 2$

$((SR).(R)) \leftarrow (X)$

PUI   R,A(X)   Push halfword immediate

$(R) \leftarrow (R) + 2$

$((SR).(R)) \leftarrow \left|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X)\right|$

PUS   R,A(X)   Push halfword from stream

$(R) \leftarrow (R) + 2$

$((SR).(R)) \leftarrow ((SX).\left|A + (X)\right|)$

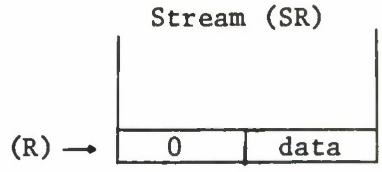
PUP   R,A(X)   Push halfword from procedure

$(R) \leftarrow (R) + 2$

$((SR).(R)) \leftarrow ((JSTRNM).\left|\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X)\right|)$

Push Byte

The contents of R are incremented by two. A sixteen-bit value consisting of a zero in the high byte and the eight-bit value defined by the second field in the low byte is placed in the stack named by SR at the location specified by the incremented value of the R.



stack after "push byte"

Pop Halfword

A sixteen-bit value is fetched from the stack named by SR at the location specified by R and stored in the location specified by the second field. Then the contents of R are decremented by two.

PO R,A(X) Pop halfword into core

$\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A) \leftarrow ((SR).(R))\ \underline{\text{else}}\ (A + (X)) \leftarrow ((SR).(R))$

$(R) \leftarrow (R) - 2$

POR R,X Pop halfword into register

$(X) \leftarrow ((SR).(R))$

$(R) \leftarrow (R) - 2$

POS R,A(X) Pop halfword into stream

$((SX).|A + (X)|) \leftarrow ((SR).(R))$

$(R) \leftarrow (R) - 2$

Pop Byte

A halfword is fetched from the stack named by SR at the location specified by R. The low eight bits of this halfword are stored in the location indicated by the second field. Then the contents of R are decremented by two.

POB R,A(X) Pop byte into core

$\underline{\text{if}}\ X = 0\ \underline{\text{then}}\ (A) \leftarrow (|(SR).(R)|_{8-15})\ \underline{\text{else}}$

$(A + (X)) \leftarrow (|(SR).(R)|_{8-15})$

$(R) \leftarrow (R) - 2$

POBR R,X Pop byte into register

$(X_{8-15}) \leftarrow (\left|(SR).(R)\right|_{8-15})$

$(R) \leftarrow (R) - 2$

POBS R,A(X) Pop byte into stream

$(\left|(SX).\left|A + (X)\right|\right|_{8-15}) \leftarrow (\left|(SR).(R)\right|_{8-15})$

$(R) \leftarrow (R) - 2$

SECTION VIII

THE CONTROL STACK

Each job is provided with a control stack which is used to hold information about the environment. A user may save the contents of a register, stream register or condition instruction by means of a "push into control stack" (PUC) instruction. The "pop from the control stack" instruction (POC) restores information entered in the control stack by a PUC.

The only other instructions which affect the control stack are the CALL and RETN instructions. These instructions are defined in the next section. CALL leaves a record of seven halfwords, called the activation record, in the control stack. This means that the control stack contains a running history of the job. Some programs are interested in using this history -- for example, debugging aids. Such programs may use the control stack as a stream. This can be done by picking up the name of the control stack from location CSNAME in the job area. The pointer to the top of the control stack is also needed. The stream page of this pointer is in $CSEXT_{0-7}$; the low part of the pointer is in $CSREG_{8-15}$.

Stack overflow and underflow are recognized for all operations on the control stack. In addition, pseudo overflow is recognized for the control stack.

1. True overflow or underflow: If during the course of a control stack instruction, true overflow or underflow occurs, then $SAVREG_0$ is turned on, the "stack overflow/underflow" bit is turned on, and the instruction becomes a NOP.

2. Pseudo overflow occurs if during the interpretation of a PUC or CALL, the last page of the control stack stream is entered. In this case $SAVREG_0$ is turned on and the "stack overflow/underflow" bit is turned on, but the execution of the instruction is continued.

In the descriptions of control stack instructions which follow, it is assumed that underflow and overflow do not occur.

CONTROL STACK INSTRUCTIONS

Push into the Control Stack

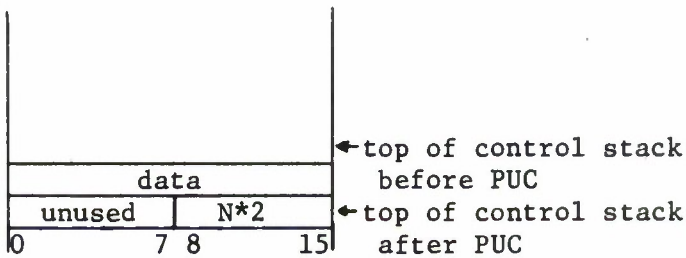
PUC N

The format of PUC is different from that of any other instruction. It is a short instruction:

0 7	8 15
5 2	N

N is the 8-bit address of the halfword in the job area which is to be saved. N must be the address of a general register, stream register or (part of a) condition instruction. These are all grouped together in the bottom of the job area. If N is not an allowable address, which is true if N $\geq$ 50, then the microprogram will not execute the PUC. It becomes a NOP.

If N is legal, first the halfword pointed to by N is pushed into the control stack. Then a control word equal to N*2 is pushed in. We have



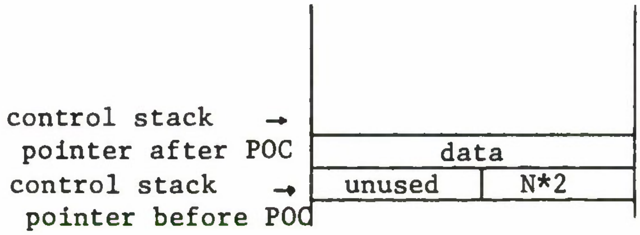
The condition code is not changed.

Pop from Control Stack

POC

The halfword in the top of the control stack is examined. If bit 15 = 1, then the top record in the control stack is an activation record (see CALL) rather than a PUC record. In this case the POC has no effect, but the carry bit in the condition code is turned on.

If bit 15 is off, then the top record in the control stack is a PUC record. This means bits 8 to 14 of the top halfword contain the address in the job area in which the data should be stored.



This data is fetched (from the next word in the control stack) and stored in the correct place. Recall that N is the address of a general register, stream register or (part of a) condition instruction. If N is the address of a stream register, then the effect of the POC instruction on N is precisely the same as if a "load stream name" instruction were being used to set the stream register at N. This means, if

$$(N) = \text{data}$$

then nothing further is done, but otherwise the page referred to by the extension of the stream register at N is <u>aged</u> and then

$$(N) \leftarrow \text{data.}$$

If N is the address of a general register or condition instruction, then

$$(N) \leftarrow \text{data.}$$

The effect of POC on the condition code is:

C	V	G	L	
1				Top record in Control Stack is an activation record.
0				Top record in Control Stack was a PUC record.

POC does not affect the V, G or L bits.

Programmer's Note: POC is defined so that a user can restore the job area without having to count the number of PUC's. Instead he performs

```
A      POC
       BFC     8,A(0)        (see next section)
```

The branch falls through when the activation record is reached.

SECTION IX

PROCEDURES

Procedures reside in streams, one procedure per stream. Procedure streams have the following format:

bytes 0 - 1	alternate name for job in job area 0.
bytes 2 - 3	alternate name for job in job area 1.
⋮	
bytes 1E - 1F	alternate name for job in job area F.
bytes 20 - 23	pointer to control information for procedure.
bytes 24 - FFFF	body of procedure; the first location of the procedure is at location 24.

An alternate name is the stream name of a procedure stream. Alternate names have the following use. Suppose a job, job n, wishes to use an alternate representation of a system procedure, say SQRT, in such a way that even system routines which it calls will use its version of SQRT. Other jobs running simultaneously may also refer to SQRT; these jobs will want the original version of SQRT. Both requirements are satisfied if job n sets halfword 2n in the procedure stream containing the original version of SQRT to the stream name of the procedure stream containing its version of SQRT. This binding may only occur at execution time and must be undone when job n stops running. A zero in halfword 2n means there is no alternate name for job n.

The pointer to control information is large enough to hold an entire stream address. What the control information is and where it is stored will be defined by software.

Jobs under Venus are almost always running from a procedure (in stream mode with $COND_{15}$ on). The name of the procedure which is running from a job area is stored in JSTRNM in that job area. When this job is the running job, the core address of the next instruction to be executed is kept in the micro-registers. When the job is not running, this pointer is kept in PC in the job area. In either case the number of the stream page containing the next instruction to be executed is kept in ICSTRP in the job area. This stream page is locked in.

In addition to this page, one other procedure page may be locked in. This is called the alternate procedure page. The alternate procedure page is the page of the procedure referenced immediately before the current page. The number of this page and the number of the core page containing it are stored in JEXT in the job area. JEXT is like any stream register extension: if the core page equals zero, this indicates no page is locked in.

The stepping of the instruction counter in stream mode may cause a page fault within the procedure stream. When the microprogram determines that a new procedure page may be required, which would happen as the result of a branch instruction, it compares the new stream page, SP, with the current stream page, SP_C. If these pages are the same, there is no difficulty. If they are different, or if the instruction counter has run over the top of the current stream page, the microprogram compares the new stream page with the alternate stream page, SP_A. SP_A is stored in JEXT:

$$(JEXT) \leftarrow \boxed{\quad SP_A \quad | \quad CP_A \quad} .$$

If

$$CP_A \neq 0 \wedge SP = SP_A,$$

then the current and alternate procedure pages are exchanged. This means:

1) $(JEXT) \leftarrow \boxed{\quad SP_C \quad | \text{ current core page }}$

2) $(ICSTRP) \leftarrow SP_A$

3) The new core location is stored in the micro-registers. No page swapping is required in this case.

If

$$CP_A = 0 \vee SP \neq SP_A,$$

then the alternate procedure page is <u>aged</u>, and the current procedure page becomes the alternate procedure page:

$$(JEXT) \leftarrow \boxed{\quad SP_C \quad | \text{ current core page }} .$$

Then the new stream page is located (and possibly fetched from the disk). Finally,

$$(\text{ICSTRP}) \leftarrow \text{SP}$$

and the new core location is stored in the micro-registers.

Two instructions, CALL and RETN, are supplied for transfer of control between procedures. Branch instructions only transfer control within a procedure.

BRANCH INSTRUCTIONS

The result of executing a branch instruction in Venus depends on the mode in which the job is running. Jobs in Venus usually run in stream mode, but occasionally a job will run in core mode. Bit 15 of COND in the job area tells what the mode is. If bit 15 in COND is off, the branch will be to a location in core.

If bit 15 in COND is on, then the job is in stream mode. The branch will be to another location in the same stream in this case; in other words, the branch goes to another location in the same procedure. Control cannot be switched from one procedure to another by means of a branch instruction.

Branch instructions do not affect the condition code. However, some branch instructions test the condition code. In these instructions the first field is a 4-bit mask, M. This mask is compared with the condition code. The four bits of the condition code are arranged:

$$\text{condition code} = \boxed{C\,|\,V\,|\,G\,|\,L}\ .$$

Therefore, M_0 is compared with C, M_1 with V, M_2 with G and M_3 with L.

Branch on True Condition

A logical "and" is performed on the 4-bit mask M specified in the instruction and the condition code. If the result is non-zero, which means at least one of the conditions being tested is on, then the branch is taken. Otherwise the next instruction is executed. The second field defines the branch address (depending on the class of the instruction). This address is interpreted according to the value of bit 15 of COND.

BTC M,A(X) Branch on True Condition

branch address = $\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$

BTCR M,X Branch on True Condition

branch address = (X)

Programmer's Note: If M = 0, then BTC and BTCR are NOPs.

Branch on False Condition

A logical "and" of the mask M and the condition code is performed. If the result is zero, which means all conditions being tested are off, then the branch is taken; otherwise the next instruction is executed. The second field defines the branch address. This address is interpreted as being in the procedure stream if bit 15 of COND is on; otherwise it is interpreted as being in core.

BFC M,A(X) Branch on False Condition

branch address = $\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$

BFCR M,X Branch on False Condition

branch address = (X)

Programmer's Note: If M = 0, then BFC and BFCR are unconditional branches.

Branch and Link

First the current instruction counter, which is a 16-bit pointer in core or within the procedure stream, depending on $COND_{15}$, is saved in R:

$$(R) \leftarrow \text{current instruction counter.}$$

Then the branch is taken to the branch address defined by the second field. The branch address is interpreted according to the value of bit 15 of COND.

BAL R,A(X) Branch and Link

branch address = $\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$

BALR R,X Branch and Link from Register

branch address = (X)

<u>Branch on Index High</u>

Prior to the execution of this instruction:

(R) = 16-bit count

(R + 1) = 16-bit increment

(R + 2) = 16-bit limit.

First of all

$(R) \leftarrow (R) + (R + 1).$

Then (R) is compared with (R + 2). If (R) $\leq$ (R + 2), the program continues in sequence. If (R) > (R + 2), then the branch is taken to the branch address specified by the second field. The branch address is interpreted according to the value of bit 15 of COND.

BXH R,A(X) Branch on Index High

branch address = $\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$

BXHR R,X Branch on Index High

branch address = (X)

Branch on Index Low or Equal

Prior to the execution of this instruction:

$$(R) = \text{16-bit count}$$

$$(R + 1) = \text{16-bit increment}$$

$$(R + 2) = \text{16-bit limit.}$$

First of all

$$(R) \leftarrow (R) + (R + 1).$$

Then (R) is compared with $(R + 2)$. If $(R) > (R + 2)$, the program continues in sequence. Otherwise, if $(R) \leq (R + 2)$, the branch is taken to the branch address specified by the second field. The branch address is interpreted according to the value of bit 15 of COND.

BXLE R,A(X) Branch on Index Low or Equal

branch address = $\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$

BXLR R,X Branch on Index Low or Equal

branch address = (X)

CALL AND RETURN INSTRUCTIONS

Call Procedure

CALL A(X)

CALL is the instruction which passes control from the present procedure to a new one. The name of the new procedure is given by the second field:

$$\text{Name} = \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|.$$

CALL has two primary tasks to perform: it must save a snapshot of the present environment in the control stack, and then it must start the new procedure. This snapshot is called the activation record:

			control stack ←pointer before CALL
(JSTRNM)			
(SR1)			
old instruction counter			
(R1)			
(R9)			
unused	job number	condition code	
condition information	condition mask		←control stack pointer after CALL
0 7	8 15		

Activation Record

The contents of JSTRNM, SR1, R1 and R9 are picked up from the job area. The old instruction counter is the stream address of the instruction after the CALL. The job number and condition code are taken from the micro-registers.

The condition information given depends on whether the CALL being executed is a condition instruction. If not,

$$\text{condition information} = \left| \left| \text{MASK} * 2 \right| \wedge \text{FF} \right| \vee 1.$$

If the CALL is a condition instruction, then the condition information comprises 8 bits of the form: zero to seven ones followed by zero to seven zeros followed by a one; for example,

11100001.

If the condition information is compared with MASK, then its left-most zero is in the position in the condition mask of the condition which caused the interrupt (in the example, "stack overflow/underflow") unless the "illegal stream instruction" interrupt is being executed. In this case, the condition information contains all ones. In addition, when the CALL is a condition instruction, the condition mask in the job area is set to

$$(\text{MASK}) \leftarrow (\text{condition information}) \wedge (\text{MASK}).$$

This means that the interrupt being honored and all lower priority interrupts are disabled for the new procedure.

The CALL instruction also turns on the "call interrupt" bit unless it is the "call interrupt" instruction.

Finally, to start the new procedure, the present procedure page and the alternate procedure page are _aged_ and $JEXT_{8-15}$ is set to zero. Then the first page of the new procedure is located (and fetched from the disk if necessary). If the n^{th} halfword of this procedure is zero, where n is the job number, then the new instruction counter is set to point to location 24 (hex) of that procedure. If the n^{th} halfword is non-zero, then it contains the name of another procedure. The first page of this procedure is located and the instruction counter set to location 24 (hex) of this procedure.

The condition code is not affected by the CALL instruction.

Return from Procedure

RETN R,A(X)

The first thing the RETN instruction does is to search down the control stack for an activation record. One word is popped from the control stack and the low bit examined to determine if it indicates return information (bit 15 = 1). If not, a PUC record has been found, and it must be skipped. The next word is also popped, bits 4 and 5 of BITS are turned on, and the RETN is interrupted to permit the channel to function. Then the search for the activation record is continued.

When an activation record is found, it is used to restore the environment at the time of the CALL. The condition mask, MASK, is restored to its previous value. The condition information is examined; if the CALL was caused by an "every instruction" interrupt, the "every instruction" bit in ON is turned off. The condition code is set to its old value. R1 and R9 are set to the values in the stack. SR1 is compared to the value in the stack; if the old and new names do not match, then the current contents of SR1 are _aged_ and the old name is restored.

SECTION X

MULTIPROGRAMMING AND SEMAPHORES

Venus is a multiprogramming system supporting up to sixteen jobs running concurrently. However, for the most part, jobs running under Venus need not be concerned with the fact that other jobs are also running. There is interaction between two different jobs only if the jobs desire it.

The question remains of how control is transferred from one job to another. Although there is a limited time-slicing capability in Venus, the primary reasons for a new job to start running are:

1. The old running job is temporarily unable to proceed, or

2. Some other job with higher priority than the running job is now able to proceed.

The mechanism which allows a job to stop running and then later starts it running again is provided by semaphores*.

Semaphores are a special kind of data provided in Venus. They are used to control the sharing of resources and to permit the synchronization of concurrent processes. One semaphore is associated with each such resource or synchronization.

A semaphore is a seven-bit signed integer counter. Attached to each semaphore is a queue. The queue hangs from an eight-bit pointer which is stored in the same halfword as the semaphore:

```
            0           7 8          15
halfword = [   pointer   |  semaphore  ] .
                         ↑
                      sign bit
```

This halfword may be located in core or in a stream.

---

* Dijkstra, E. W., "The Structure of the 'THE'-Multiprogramming System", <u>Communications of the ACM</u>, Vol. 11, No. 5; May 1968.

If the value of a semaphore is non-negative, then the attached queue is empty. In this case the pointer equals 1. If the semaphore is negative, then the queue is represented by a chain starting at the attached pointer and going through the LINK registers in the job areas. The pointer equals the core page number of some job area. In this job area the LINK register either equals 1, meaning the queue stops there, or it equals the page number of some other job area. In addition, the absolute value of the semaphore equals the number of jobs on the queue.

Only two instructions may be used to manipulate semaphores. These are the P and V instructions, also defined by Dijkstra. An example of the use of P and V for controlling sharing of a resource is given here. Suppose the console teletype is shared among the jobs. If a job wishes to use it, it calls a system program. Before using the console teletype, this program performs a P on the semaphore controlling its use. By the time the instruction following the P is executed, the program is assured that it is the sole owner of the console teletype. It then proceeds to use it for a limited time (say one message and one response) and, when finished, releases it by performing a V on the same semaphore.

A semaphore used to control the sharing of a resource is called a public semaphore by Dijkstra. Such a semaphore should be initialized to 1 (and its attached pointer also to 1). A job which performs a P on a public semaphore must later perform a V on it.

Semaphores used for synchronization are called private semaphores by Dijkstra. They are initialized to either 0 or 1 (and the attached pointer initialized to 1). If a semaphore is initialized to 0, then a V must be performed on it before a P can be concluded on it. Private semaphores differ from public ones in that different jobs may perform the P and V. The most common use of private semaphores is to synchronize a job with its I/O.

Jobs in Venus are always in one of three states. A job is either:

1. The running job.

2. Unable to run. In this case it has performed a P on some semaphore and is waiting for a V.

3. Ready to run. In this case it is waiting on the JOBSEM (location A0). The JOBSEM is organized just like a semaphore, but P's and V's must never be performed on it.

Each job in Venus has a priority. This priority is assigned by software and stored in PRIOR in the job area. Whenever the microprogram must choose which job to remove from a queue, it will select the one with highest priority; if several jobs have the same priority, then the one which has been on the queue the longest is selected.

Two operations are used to describe P and V.

1. "Job J is added to the queue" means

```
begin
   (J*1000 + LINK) ← (pointer);
   (pointer) ← J*10
end
```

2. "A job J is unqueued from the nonempty queue" means

```
begin
   J ← (pointer_0-3);
   L ← J;
   for (L*1000 + LINK) ≠ 1 do begin  L ← (|L*1000 + LINK|_0-3);
      if (L*1000 + PRIOR) ≥ (J*1000 + PRIOR) then J ← L
      end
end
```

P AND V INSTRUCTIONS

P and V almost completely define the flow of control from job to job in Venus. V often has the effect of adding a job to the JOBSEM. P may cause the number of jobs on the JOBSEM to be reduced by one. When there are no jobs on the JOBSEM and a P is performed on a non-positive semaphore, the microprogram enters the idle loop. P and V do not change the condition code.

P of Semaphore

The semaphore specified by the second field is decremented by 1:

(semaphore) ← (semaphore) - 1.

If the result is non-negative, control proceeds to the next instruction.

If the result is negative, two things happen. First, the job is suspended and added to the queue attached to the semaphore (operation 1). Next the microprogram looks for a new running job. If the JOBSEM is empty then the idle loop is entered. Otherwise a job is unqueued from the JOBSEM (operation 2) and becomes the running job.

The two versions of this instruction differ only in the computation of the address of the halfword containing the pointer and the semaphore.

P   A(X)   P of semaphore in core

address = $\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$

PS   A(X)   P of semaphore in stream

address = $(SX).\left| A + (X) \right|$

V of Semaphore

The semaphore specified by the second field is incremented by one:

$$(\text{semaphore}) \leftarrow (\text{semaphore}) + 1.$$

If the result is positive the instruction is complete.

If the result is non-positive, a job, J, is unqueued from the queue attached to the semaphore (operation 2). If the priority of J is greater than the priority of the running job, then the running job is added to the JOBSEM (operation 1) and J becomes the running job. Otherwise J is added to the JOBSEM (operation 1).

The two forms of V differ only in the computation of the address of the halfword containing the pointer and semaphore.

V   A(X)   V of semaphore in core

address = $\left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$

VS   A(X)   V of semaphore in stream

address = $(SX).\left| A + (X) \right|$

SECTION XI

THE MICROPROGRAMMED MULTIPLEX CHANNEL

Input/Output in VENUS is handled primarily by means of the microprogrammed channel. The channel is started by execution of a "start I/O" instruction (SIO or SIOR). These instructions will be described in the next section. MSW_4 (location A6) must be on if the channel is to run.

A job in VENUS may run simultaneously with its I/O. After giving the "start I/O" instruction it continues to run. When it needs to know that its I/O is finished, it performs a P on its IOSEM (located in the job area). The normal sequence of instructions is like:

```
SIO     R,A(X)
 .
 .
 .
JOBA    R            R ≠ 0

P       IOSEM(R)
```

When the job reaches the instruction after the P, this means some other process has performed a V on the IOSEM. If the job is only running one device, then it can assume the I/O is finished. If it is running more than one device, then it must determine which device performed the V (by consulting TSTAT in the DSW of the device -- explained later).

The "start I/O" instruction causes the execution of a program of channel commands. This program is located in core, and is responsible for initializing the Device Status Word (DSW) for the device on which I/O is to be performed. For example, it initializes the (in-core) location of the buffer, the number of bytes to be transferred, and the type of I/O to be performed. Much of the initialization is device-dependent. Finally, the channel-command program starts the I/O and transfers control to the channel.

Between the execution of the instructions, the microprogram checks to see if any I/O device requires attention (see Figure 3). If any does, the channel starts to run. The channel will handle the I/O transfer according to the information in the DSW.

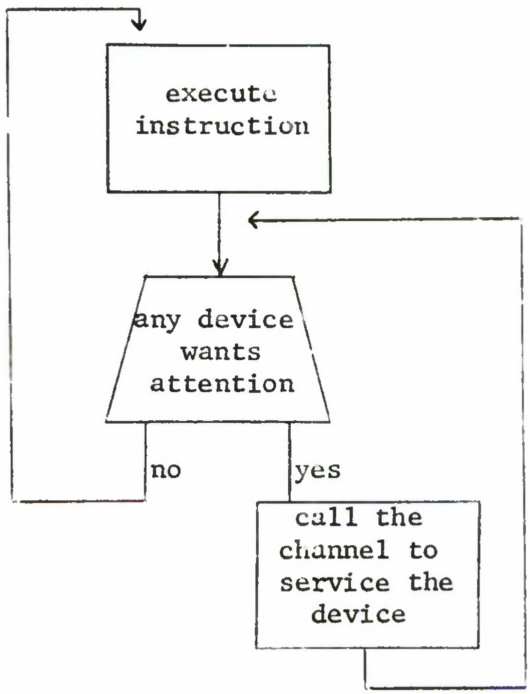


Figure 3. How the Channel Fits into the Microprogram

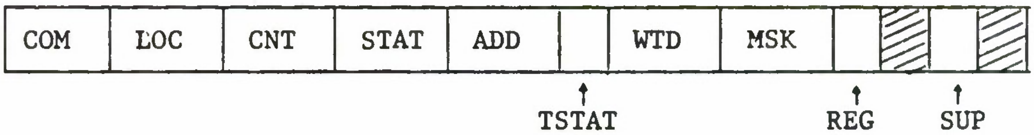
When the channel determines that the transfer is complete, it does one of two things: either it performs a V on the IOSEM of some job or it returns to the channel-command program at the instruction after the one which transferred control to the channel.

The DSWs are stored in the DSW table (in-core). This table is indexed by device number. A pointer to the start of this table is stored in NOVLIS (B0). Each DSW contains 16 bytes of information.

16 bits	COM	contains the core address of the next channel command.
16 bits	LOC	contains an address in the buffer.
16 bits	CNT	contains the number of bytes to be transferred.
16 bits	STAT	The status of the device at each attention is 'or'ed into the left byte and its complement into the right byte if the channel is instructed to do so.
16 bits	ADD	contains the core location of a 16-byte table; each bit in the table corresponds to one of the 128 7-bit ASCII characters. Input characters may be checked to see if the corresponding bit in the table is set. This option can be used when reading to terminate I/O or to skip characters.
4 bits	TSTAT	Transfer status, which has the format:

bit 0	INUS	is on if the device is in use.
bit 1	SPUR	is on if a spurious attention has occurred (attention requested while INUS = 0).
bit 2	VR	is on if the last transmission for this device ended with a V on the regular job and a spurious interrupt did not occur.
bit 3	VS	is on if the last transmission for this device ended with a V on the supervisor.

12 bits	WTD	What To Do, which actually controls the transfer (described later).
16 bits	MSK	MSK may be compared with the status during a data transfer and the transfer terminated if a match is detected.
4 bits	REG	The job number of the job controlling the device (the regular job).
4 bits	SUP	The job number of the supervisor. The supervisor job number is set by software.



The DSW

THE CHANNEL-COMMAND PROGRAM

At the time the "start I/O" instruction is given, the number of the job giving it is stored in REG. This is the only automatic initialization done by the microprogram. The channel commands must initialize the DSW so that the data transfer may proceed correctly. Channel commands have the following format:

bits 0 - 2	opcode
bit 3	X
bits 4 - 15	WTDC - The What To Do in the Command
bits 16 - 31	A

All eight possible opcodes are recognized. They are:

LCNT	000	A is stored in CNT. This command initializes the count of the number of bytes to be transferred.
LADD	001	A is stored in ADD. This command initializes the start of the ADD table (in-core). The ADD table is only used when the type of transfer is READ (explained later).
LLOC	010	A is stored in LOC. This command initializes the location of the buffer (in-core).
LMSK	011	A is stored in MSK. This option is used if the I/O transfer should be terminated based on status as well as on CNT = 0.
CHOC	100	The right byte of A is sent to the device as a command. This command starts the device going. The value of A is device-dependent.
BRCH	101	An unconditional branch in channel to A, which must be a core location.
BNZC	110	If CNT $\neq$ 0, the next command is taken at A; otherwise the program continues in sequence. A must be a core location.
TEST	111	A is compared with STAT. If A $\wedge$ STAT = 0, then the next command is skipped; otherwise the next command is taken.

Some combination of the first four commands is given to initialize the DSW. Then CHOC is used to start the device going. At about the time that CHOC is given, the channel-command program will have done as much as it can prior to data transfer. Therefore, it is necessary to discontinue running the channel-command program, while leaving information about the transfer for the channel. This is accomplished with the X bit and the WTDC. The WTDC contains all information about the kind of transfer, the reasons for termination, and the type of termination. If X is on, the WTDC will be stored in WTD in the DSW, and the interpretation of commands will be stopped. TSTAT in the DSW is set to 8 (INUS on). At this point STAT will be cleared if LAST (in WTDC) is off.

When the channel terminates a transfer, it returns control either to a job (through a V), or to the channel-command program. This permits the channel to read many pieces of data without communicating with the job. Two of the channel commands can be used to see why the previous transfer terminated (BNZC and TEST). The channel-command program will perform a V if X is off and one of the V bits in the WTDC on. The V may be performed on the IOSEM belonging to either the regular job or the supervisor. Otherwise the channel-command program can go on to initiate the next transfer. Figure 4 diagrams the execution of channel commands.

THE CHANNEL

Each time the channel is entered because a device wants attention, it first checks the INUS bit in C. If this bit is off, then a spurious attention has occurred. SPUR is turned on; if VS is on it is unchanged, but VR is turned off:

$$(\mathrm{TSTAT}) \leftarrow \left|(\mathrm{TSTAT}) \wedge 1\right| \vee 4.$$

If INUS is on, the channel decides what to do by consulting the WTD stored in the DSW for the device. The WTD has the following format:

bit 0	VREG	If this bit is set, when the I/O transfer is complete, the channel will perform a V on the IOSEM of the job controlling the device.
bit 1	VSUP	If this bit is set (and VREG is off), when the I/O transfer is complete, the channel will perform a V(IOSEM) of the supervisor.
bit 2	SIFM	Stop if match; only meaningful if CODE (below) is READ.
bit 3	MATCH	Only meaningful if CODE is READ.
bit 4	OR	Only meaningful if CODE is READ.
bit 5	LAST	If this bit is off, the status of the device and its complement will be 'or'ed into STAT each time the device requires attention.
bit 6	NSNC	No store, no count. Only meaningful if CODE is READ.

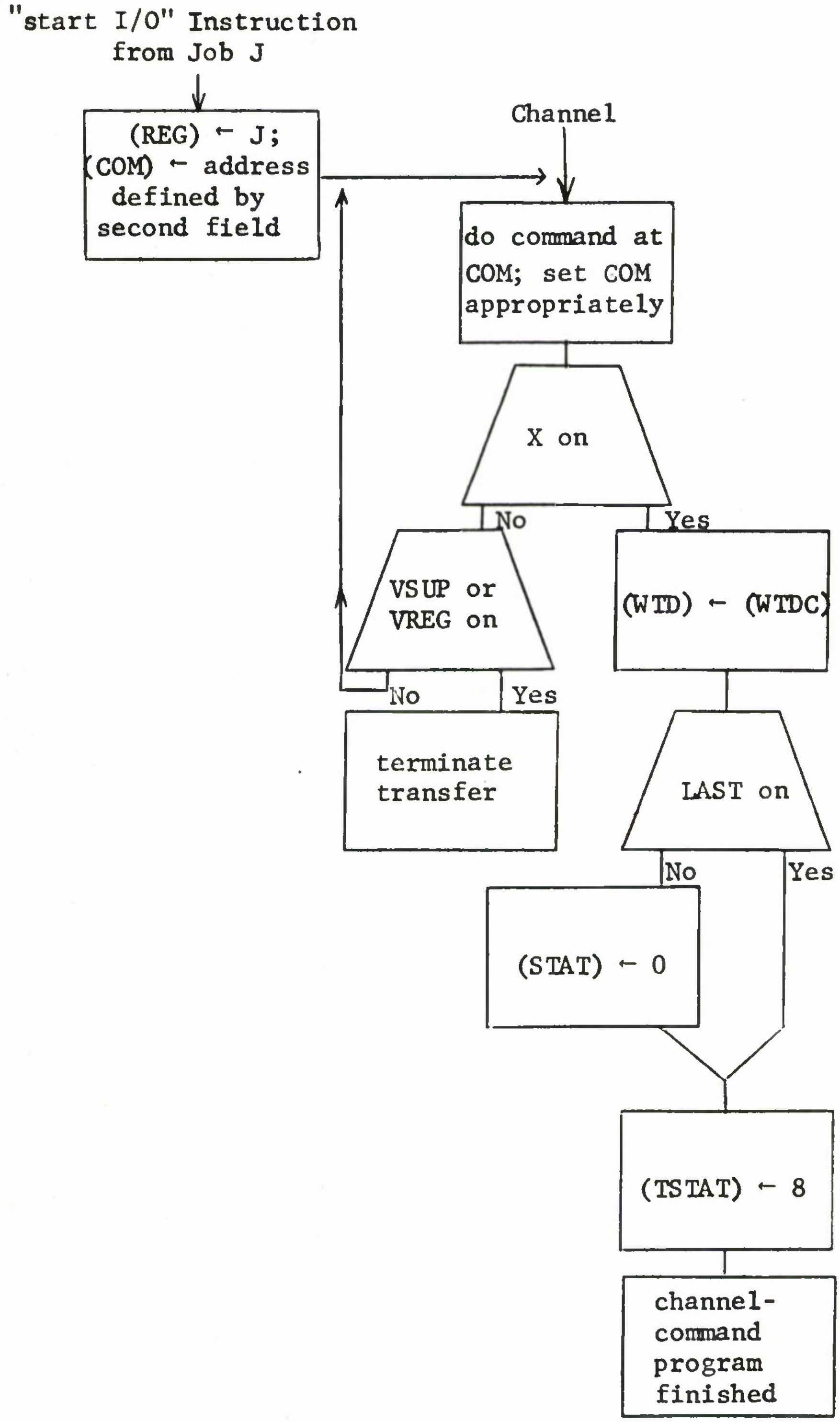


Figure 4. Execution of Channel Commands

bit 7	STFM	Store if match. Only meaningful if CODE is READ.
bits 8, 9	CODE	CODE tells the channel what type of data transfer is taking place. It will be described below.
bit 10	STNM	Store if no match. The meaning of this bit depends on the value of CODE.
bit 11	EOM	End of message. The meaning of this bit depends on the value of CODE.

0	1	2	3	4	5	6	7	8,9	10	11
VREG	VSUP	SIFM	MATCH	OR	LAST	NSNC	STFM	CODE	STNM	EOM

The WTD

Types of Termination

Three types of termination of a particular I/O transfer are available to the channel (see Figure 5). If VREG is on, it performs a V on the IOSEM of the regular job. TSTAT is set to 2 (VR on). If VREG is off and VSUP is on, it will perform a V on the IOSEM of the supervisor and set TSTAT to 1 (VS on). If both VREG and VSUP are off, the channel transfers control to the channel command in the location stored in COM in the WTD. This will be the command following the one in which the X bit was on.

Reasons for Termination

There are four reasons for termination.

1. CNT goes to 0. This reason for termination is checked automatically.

2. The device gives a status, S, such that

$$\left|\left|(MSK_{0-7}) \wedge S\right| \neq 0\right| \vee \left|\left|(MSK_{8-15}) \wedge \overline{S}\right| \neq 0\right|,$$

where $\overline{S}$ means the one's complement of S. The transfer can terminate for this reason only if the EOM bit is on in the mask, and the type of transfer is not CLOCK.

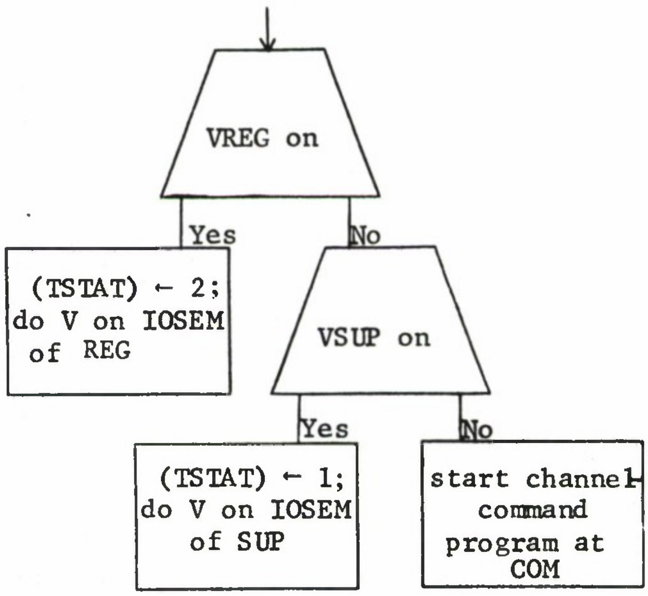


Figure 5. Terminate Transfer

3. A terminal character is read. A transfer may be terminated for this reason only if the type of transfer is READ. The definition of terminal characters is given by the ADD table.

4. Transfer termination on the first interrupt will occur if the type of transfer is CLOCK and EOM is on.

Types of Transfers

The type of transfer is indicated by the value of CODE in the WTD. Prior to the interpretation of CODE, if LAST is off, the status of the device will be 'or'ed into the left byte of STAT and its one's complement into the right byte of STAT.

CLOCK (00): A device which is used as a clock may either be used for time-slicing or to wake up a job after some number of ticks. Only the EOM bit and the STNM bits mean anything for this type of transfer.

If EOM is on, the transfer is automatically terminated without doing anything else (reason 4).

If EOM is off, the first thing the channel does is to increment a counter by 1. This counter is comprised of 3 bytes; LOC points to the leftmost byte.

Next CNT will be decremented by 1. If the result is non-zero, the interrupt is finished. If CNT = 0 and STNM is off, time-slicing will be performed. This means the interval is reset from TIMCNT,

$$CNT \leftarrow TIMCNT,$$

and the time-slicing routine is entered. This action does not constitute a termination of transfer, although the interrupt is finished. If CNT = 0 and STNM is on, the transfer will be terminated (reason 1).

Prior to interpreting the other three CODES, if the EOM is on the channel will compare the device status with MSK. If any bits match, it terminates the transfer (reason 2).

WRITE (01): The channel transfers the next byte from the buffer to the device. It increments LOC and decrements CNT. When CNT goes to 0, the transfer is terminated (reason 1).

READ2 (11): This CODE is supplied to handle the card reader, which produces two bytes at once. Two bytes are read from the device. They are stored in the buffer if STNM is on. LOC is incremented and CNT decremented after each byte; when CNT goes to 0, the transfer is terminated (reason 1).

READ (10): READ is the most complicated CODE. It has an option which allows the user to terminate the transfer on the recognition of specified characters. The user may also skip specified characters (read them without putting them into the buffer). Both of these features make use of the ADD table and are enabled by the MATCH bit. In addition, characters which are stored in the buffer always have their first bit 'or'ed with OR. This permits the user to receive a consistent 8-bit ASCII representation of a character even from a device (like a teletype) which uses bit 0 as a parity bit.

The first thing READ does is to read a character, and terminate on status if indicated. Otherwise, if MATCH is on, the bit in the ADD table corresponding to the character just read is fetched. The ADD table contains 128 bits or 8 halfwords of information. The character just read is broken into 2 hexadecimal digits. The left digit (with high order bit ignored) tells the halfword. The right digit tells the bit within the halfword. For example, if the character is a "carriage return" (8D), then bit D (i.e., bit 13) of halfword 0 of ADD is used. If this bit is on, a match is considered to have occurred. The other bits in the WTD all refer to this match:

STFM - store if match

STNM - store if no match

SIFM - stop if match

NSNC - no store, no count

If a match occurred, then STFM controls the handling of the character; otherwise STNM is in control. Three options are possible:

1. Store character just read. The character just read, C, (with bit 0 'or'ed with OR) is stored in the location indicated by LOC. LOC is increased by one and CNT is decreased by one. This case occurs if:

$$\left| \mathrm{MATCH} \wedge \mathrm{ADD_C} \wedge \mathrm{STFM} \right| \vee \left| \neg \left| \mathrm{MATCH} \wedge \mathrm{ADD_C} \right| \wedge \mathrm{STNM} \right|$$

2. Skip character just read but decrement CNT. The only action is that CNT is decreased by one. This occurs if:

$$\neg\,\mathrm{NSNC} \wedge \left| \left| \mathrm{MATCH} \wedge \mathrm{ADD_C} \wedge \neg\,\mathrm{STFM} \right| \vee \left| \neg \left| \mathrm{MATCH} \wedge \mathrm{ADD_C} \right| \wedge \neg\,\mathrm{STNM} \right| \right|$$

3. Skip character just read and do not decrement CNT. No action is taken. This occurs if:

$$\mathrm{NSNC} \wedge \left| \left| \mathrm{MATCH} \wedge \mathrm{ADD_C} \wedge \neg\,\mathrm{STFM} \right| \vee \left| \neg \left| \mathrm{MATCH} \wedge \mathrm{ADD_C} \right| \wedge \neg\,\mathrm{STNM} \right| \right|$$

Two reasons for termination are recognized for READ (in addition to terminate on status). They are: CNT goes to 0 (reason 1) and recognition of terminal character (reason 3). Transfer will terminate if

$$\mathrm{CNT} = 0 \vee \left| \mathrm{MATCH} \wedge \mathrm{ADD_C} \wedge \mathrm{SIFM} \right| .$$

Figure 6 is a flow chart of channel device-interrupt handling.

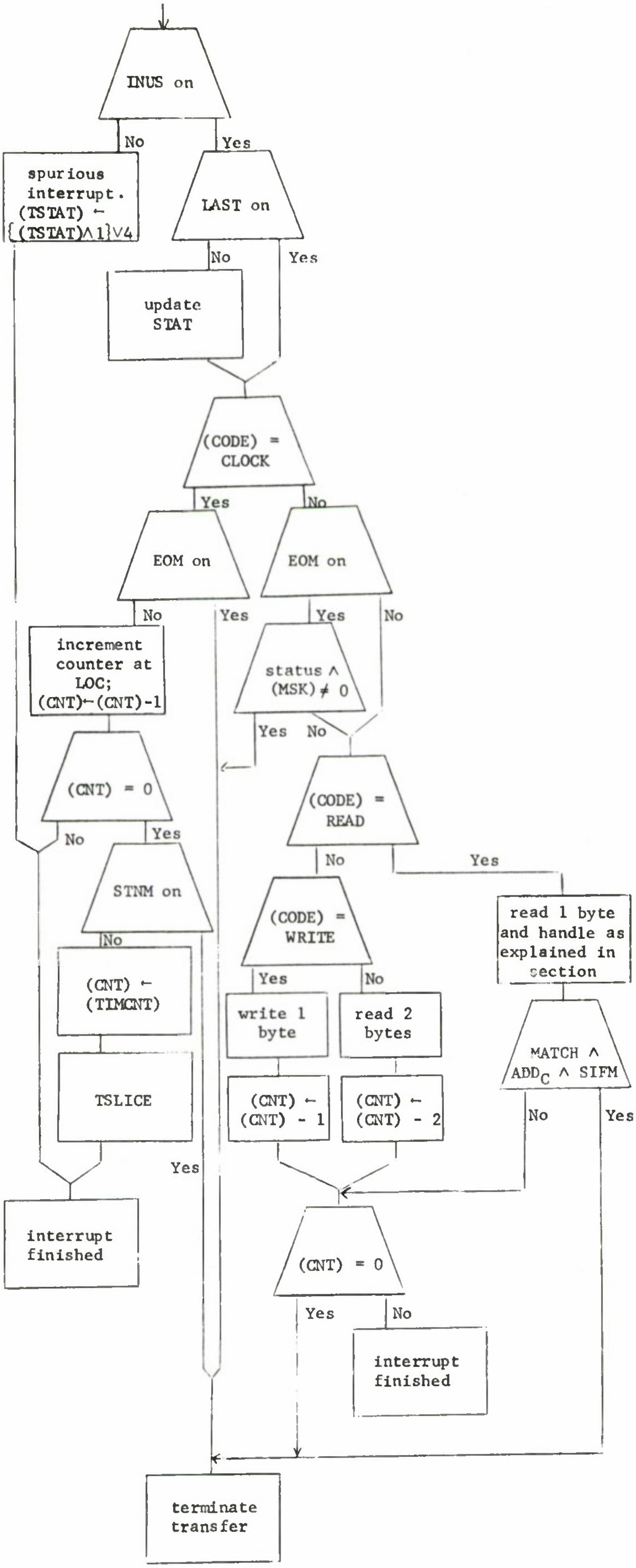


Figure 6. Channel Interrupt-Handling for a Device

SECTION XII

INPUT/OUTPUT

Input/Output in Venus is handled primarily by the channel, which is started by a "start I/O" instruction (SIO or SIOR). In addition, Venus supports a few other I/O instructions. Some devices are started by these other instructions before the "start I/O" instruction sets the channel in motion. In general, non-channel I/O should not be used. It is not truly compatible with Venus.

CHANNEL INPUT/OUTPUT INSTRUCTIONS

The second field defines the core address of the first command in a channel-command program. The two versions of the instruction differ only in the computation of this address. R contains the number of the device on which the I/O is to be performed:

$$\text{device} = (R)_{8-15}.$$

SIO R,A(X) Start I/O

$$\text{address} = \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right|$$

SIOR R,X Start I/O from register

$$\text{address} = (X)$$

The condition code is not affected by these instructions.

NON-CHANNEL INPUT/OUTPUT INSTRUCTIONS

Read Data

The right byte of R contains the number of the device being addressed:

$$\text{device} = (R_{8-15}).$$

One byte is read from this device and stored in the location which is defined by the second field. The condition code becomes:

C	V	G	L	
0	0	1	0	Device responded.
0	1	1	0	Device did not respond.

RD R,A(X) Read data into core

(<u>if</u> X = 0 <u>then</u> A <u>else</u> A + (X)) ← (device)

RDR R,X Read data into register

$(X_{8-15}) \leftarrow$ (device)

$(X_{0-7}) \leftarrow 0$

<u>Write Data</u>

The right byte of R contains the number of the device being addressed:

$$\text{device} = (R_{8-15}).$$

The second field defines an 8-bit value. This value is written to the device being addressed as data. The condition code becomes:

C	V	G	L	
0	0	1	0	Device responded.
0	1	1	0	Device did not respond.

WD R,A(X) Write data from core

(device) ← |<u>if</u> X = 0 <u>then</u> (A) <u>else</u> (A + (X))|

WDR R,X Write data from register

$\text{(device)} \leftarrow (X_{8-15})$

Output Command

The right byte of R contains the number of the device being addressed:

$$\text{device} = (R_{8-15}).$$

The second field defines an 8-bit value which is sent to the device as a command. The condition code becomes:

C	V	G	L	
0	0	1	0	Command has been sent.
0	1	1	0	Device did not respond.

OC R,A(X) Output command from core

(device) ← $|$ if X = 0 then (A) else (A + (X)) $|$

OCR R,X Output command from register

(device) ← (X_{8-15})

Sense Status

The right byte of R contains the number of the device being addressed:

$$\text{device} = (R_{8-15}).$$

The status of this device is read and stored in the location specified in the second field. The condition code is replaced by the 4 low bits of the status:

C	V	G	L	
0	1	1	0	Device did not respond.
1	0	0	0	Device busy.
0	0	1	0	End of medium.
0	0	0	1	Device unavailable.

SS R,A(X) Sense status into core

(if X = 0 then A else A + (X)) ← status

SSR R,X   Sense status into register

$(X_{8-15}) \leftarrow$ status

$(X_{0-7}) \leftarrow 0$

SECTION XIII

LEVEL 1

Streams in Venus are virtual memories. The microprogram performs the mapping between stream addresses and core address. Furthermore, streams are paged in Venus between core and the disk. Streams are divided into 256-byte pages and so is core memory, so that one stream page fits in one core page. In the course of converting a given stream address into a core address, the microprogram may discover that the desired stream page is not in core. This can happen at different places during the execution of an instruction. In this case, the microprogram starts a software routine, the Page Fault Handler (PFH), which will fetch the desired page from the disk. When finished, the PFH returns to the microprogram at the place where the page fault was discovered -- that is, into the middle of the execution of an instruction.

The PFH must run in core rather than in stream; obviously its running must cause no page faults. Its first instruction is at PGFLT (location 109E). It makes use of an in-core table, the Core Page Table (CPT), to determine which core page should be assigned to hold the new stream page. It also uses the selector channel and the disk to move pages between core and the disk. Thus it owns two resources:

1. Core Page Table

2. Selector channel (and disk).

Software functions other than page-fault handling also must be performed on the resources of the PFH (for example, creating a stream). All the functions using these resources are grouped together and comprise the <u>level 1 programs</u>. Level 1 programs as a group are non-reentrant; entry into level 1 is controlled by the DSKSEM. Level 1 programs other than the PFH are entered by means of the EL1 instruction; they start at LEVEL1 -- a different entry point.

A level 1 program runs from the job area of the job which caused it to start. The state of the micromachine is saved before the level 1 program starts running; the level 1 program preserves the state of the job area. Thus, a level 1 program can run without disturbing the environment of the job at all.

Two instructions, UNQP and DIE, are used by level 1 programs when they wish to stop running. <u>These instructions may only be used by level 1 programs</u>. In addition, the ICOR instruction is used by level 1 programs.

LEVEL 1 INSTRUCTIONS

<u>Enter Level 1</u>

EL1 R,A(X)

The second field defines a value:

$$\text{value} = \left| \underline{\text{if}}\ X = 0\ \underline{\text{then}}\ A\ \underline{\text{else}}\ A + (X) \right| .$$

This value is an argument which tells which level 1 program should be performed; R may hold an argument for this program or may return a value from this program. After the value of the second field has been computed, the state of the micromachine is saved in the job area in MICROSAVE. Micro-registers R3 and R7 contain the value; micro-register R4 contains R. MASK is also saved in MICROSAVE; then MASK is set to 0 (all conditions are disabled). The current value of the instruction counter is saved (micro-registers R0 and R1), and the instruction counter set to LEVEL1 (location 10A2).

Next a P is performed on the DSKSEM. When this P is complete, the priority of the job is raised:

$$(\text{PRIOR}) \leftarrow (\text{PRIOR}) + 80.$$

This gives the level 1 program the highest priority of all running jobs, and therefore it becomes the running job immediately. Execution begins at location 10A2.

When level 1 is entered through a page fault, the micromachine is saved as described above, but the instruction counter is set to PGFLT (109E), micro-registers R5 and R6 contain the name of the desired stream, R7 contains the number of the desired stream page, and the value of R4 saved in TR4R5 tells which stream register caused the page fault. Then the P on the DSKSEM is performed as described above.

UNQP

UNQP is the instruction which a level 1 program uses when it is finished and wishes to return control to the program which caused it to run. UNQP is a short instruction which uses neither field.

First the priority of the running job is reduced to its previous value:

$$(PRIOR) \leftarrow (PRIOR) - 80.$$

Then bit 5 of BITS is turned on to indicate that a level 1 program is returning.

If the queue hanging on the DSKSEM is not empty, a job is unqueued from this queue. The priority of this job is raised and it becomes the running job (as explained in EL1). The old running job is added to the JOBSEM.

If the queue hanging on the DSKSEM is empty, the new priority of the running job is compared with the priorities of jobs on the JOBSEM queue. If no job on this queue has higher priority than the running job, it continues to run. Otherwise, a job is unqueued from the JOBSEM and becomes the new running job. The old running job is added to the JOBSEM.

At the time at which the old running job is permitted to continue running, the micro-registers are restored from MICROSAVE and MASK is set to its old value. The microprogram continues execution of the instruction which caused the level 1 program to start running.

UNQP may be used only by a level 1 program.

DIE

DIE is performed by a level 1 program when it wishes to stop running without returning control to the program which caused it to run. DIE is a short instruction which uses neither field.

If the queue hanging on the DSKSEM is not empty, a job is unqueued from the queue. The priority of the job is raised and it becomes the running job.

If the queue hanging on the DSKSEM is empty but the JOBSEM queue is not empty, a job is unqueued from the JOBSEM and becomes the running job. Otherwise the idle loop is entered.

DIE may be used only by a level 1 program.

In Core

ICOR   A(X)

ICOR is the instruction used by a level 1 program when it wishes to discover whether a stream page is in core.

If the stream page containing (SX).{A + (X)} is in core, the extension of SX is set accordingly and the C bit in the condition code is turned off. If the page is not in core, the C bit is turned on. In either case, the previous reference in the extension of SX is aged if necessary. The V, G and L bits are all turned off.

ICOR is primarily of use to level 1 programs. It permits them to make use of the microcode to determine if a page is in core without triggering a page fault.

When the micromachine discovers that the running job must stop running, it does the following. CORET1 and $CORET2_{0-7}$ comprise a 3-byte counter which is being updated by the clock. The clock only gives an interrupt when it exceeds more than 8 bits of data. Thus the current value of the clock (accurate to 1 millisecond) may be considered the fourth byte of current time. The microprogram forms this 4-byte counter CTIME and then:

TIME ← TIME + CTIME - CURRT	This increments running time of job.
CURRT ← CTIME	This sets time of last job change.

In these computations, 4-byte arithmetic with carries is being performed.

Whenever the microprogram leaves the idle loop it performs the same calculations using IDLET rather than TIME:

IDLET ← IDLET + CTIME - CURRT	This increments idle time.
CURRT ← CTIME	

Thus, jobs will never be charged for idle time.

BOOTS*

BOOTS is a simple microprogrammed core loader for the I-3. BOOTS is capable of loading either punched paper tape from the teletype (or possibly from a high-speed reader) or cards. It will perform scatter-loads to any part of real memory. A continuous display of the address currently being loaded and the loaded contents thereof is generated.

---

* The section on BOOTS is an update of previous work by J. E. Sullivan.

Formats

Tape

The tape expected by BOOTS is a standard 8-channel paper tape. A byte, e.g., "E6" (hexadecimal), is punched HHHNN/HHN where H = "hole", and the slash represents the sprocket feed holes.

The tape consists of any number of "records", each preceded by an arbitrary amount of blank tape (sprockets only). A record is punched as follows:

EE

start address left

start address right

data

.
.
.

EE

F4 or F8

The left and right parts of the start address specified become the first (16-bit) address loaded by the specified data; data beyond the first two bytes are, of course, loaded into successive locations. Two points should be noted: (1) the start address must be even (a halfword address), and (2) an even number of data bytes (an integral number of halfwords) must be given in the data -- otherwise, the last byte is simply lost.

In the "data" section, the data byte "EE" is always represented by the two-byte sequence

EE
FØ

This combination counts as one byte in rule (2) above.

The terminal code "F4" signals the end of the current record; the loader immediately begins scanning for the next record. The code "F8" (or equivalently, "FF") has a similar meaning except that the processor halts before resuming the scan. This is normally the terminal code of the "last" record on a tape.

Cards

A byte is read from a single card column in the following format:

Bits	Rows
0 - 3	12 - 1
4 - 7	6 - 9

Rows 2-5 should not be punched on any card read by BOOTS.

Records have exactly the same format as for paper tape (above). However, one and only one record may appear on a single card. (The record may begin and end anywhere on the card.) Unlike tape, a record terminated by an F8 code will cause reading of the input device to cease as well as the processor operation.

Operating Procedures

Tape

1. The TTY should be on-line, the mode switch on "KT", the tape reader toggle switch on "STOP".

2. Turn the processor mode switch to "MEMW" and put 02 in switches 8-15.

3. Initialize.

4. Execute.

5. Put the tape to be loaded in the reader. Flip the toggle to "RUN". The display should be active while the tape is loading. The processor (though not the tape) will halt (wait light on) when a EE/F8 or EE/FF terminal code is sensed.

6. Flip the toggle to "STOP". To load another tape, go back to Step 4.

Cards

1. The card reader should be ready-to-go, with the "motor" and "start" buttons on and the deck to be loaded in the input hopper.

2. Turn the processor mode switch to "MEMW" and put 04 in switches 8-15.

3. Initialize.

4. Execute. The display should be active while the deck is loading. The reader and processor will stop (wait light on) when a record containing a EE/F8 (or EE/FF) code is sensed.

   If the card hopper should become empty or the reader otherwise leave the "ready" state, loading will resume automatically when the reader is returned to the "ready" state as defined in (1) above.

5. To load another deck, go back to Step 4.

DISPLAY PANEL

Display Lights

There are two rows of 16 lights each on the display panel. They are used to display three types of information:

1. Normal display. The upper lights contain the core instruction counter. The lower lights contain the number of the current stream page in positions 0-7, the current job number in positions 8-11, and the condition code in positions 12-15.

2. Memory display. The upper lights contain the core memory address and the lower ones contain the contents of that memory location.

3. BOOTS display. This is a memory display produced for each location loaded by the BOOTS loader.

Power

The lower of the three push button switches at the lower left of the display controls power. The light next to it is lit whenever power is on. Power can be turned off whenever the machine is operating normally. Power down causes the current core instruction counter to be stored at PROR1 (AA) and the job number and condition code at location PR2(AC).

Initialize

The button immediately above the power button is the initialize button. Pushing this button initializes all I/O interfaces and the computer. In addition, the computer starts running at the beginning of its microprogram. The microprogram tests the upper rotary switch to determine what it will do. There are four meaningful positions:

1. OFF. This operates the BOOTS loader. The number of the input device is placed in the right 8 of the row of 16 push-button switches below the display lights. The initialize and execute buttons are pushed and the device operated. After loading is complete, the loader may be run again by pushing the execute button. This button is directly above the initialize button.

2. Register Display (RD). The instruction counter, job area number, and condition code are restored to what they were before the last power down (from PROR1 and PR2), and the computer enters the display loop.

3. INST. The instruction counter is restored to what it was before the last power down and the job area number and condition code are set to 0. The computer then enters the display loop.

4. PSW. The instruction counter is restored from the last power down. The job area number is taken from switches 8-11; the condition code is taken from switches 12-15. The computer then enters the display loop.

Display Loop

Between the execution of instructions, at the same time that it checks for devices requiring attention, the microprogram checks whether a display is being requested. A display is requested by pushing the execute button with the lower rotary switch set to other than RUN, or by pushing the initialize button with the upper rotary switch set to RD, INST, or PSW. Such an action causes the microprogram to enter the display loop. While it is in the display loop, the wait light above the power light is lit. The microprogram will resume execution of instructions when execute is pushed with the lower rotary switch set to RUN or VARI.

The following functions may be performed while in the display loop:

Reading and Writing Memory

Pushing the execute button with the lower rotary switch straight down (MEMA) causes the memory location inserted in the switches to be displayed. After that, whenever the execute button is pushed and the lower rotary switch is set at MEMR the next higher location is displayed. With the rotary switch at MEMW, the location being displayed is changed to the contents of the switches and the next higher location is displayed. MEMR and MEMW may be used in any sequence desired.

The Starting Address

When the lower rotary switch is set at ADRS, pushing the execute button transfers the contents of the switches to the instruction counter. makes the normal display, and enters the display loop.

Stopping

When the lower rotary switch is set at HALT, pushing the execute button causes an entry into the display loop and a normal display.

Running

When the switch is at RUN, pushing the execute button causes a normal display and, if there is a running job, it is caused to run. A running job exists if the instruction counter is greater than FF. Otherwise the machine is in the idle loop.

Single Stepping

When the lower rotary switch is set to VARI, pressing the execute button causes the machine to enter the single state. In this state, if the knob below the upper rotary switch is in any position but SNGL, the machine will operate instructions at a slow rate making the normal display between them. The speed is controlled by the knob. If the knob below the upper rotary switch points to SNGL, the microprogram executes one instruction, makes the normal display, and enters the display loop. The machine will not leave the single state until the execute button is pushed with the lower rotary switch in some position other than VARI. Single stepping will take place only if MSW_4 (location A6) is set.

THE IDLE LOOP

When the running job performs an instruction which causes it to stop running (for example, a P instruction), and there is no job ready to run (the JOBSEM queue is empty), then the microprogram enters the idle loop. It does this by setting the instruction counter to a value in core page zero and then entering the display loop. It will remain in the display loop until the channel performs a V, or until a new job is entered through the display.

APPENDIX I

OPCODE MATRIX

The opcode of an instruction consists of the row number in hex followed by the column number in hex. Thus, LH is B9. Instructions in rows 2, 5, 8 and C are short. All others are long.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1	P	V	STB	POB	STH	PO	SSN					OC	RD	WD	SS	
2			STBR	POBR	STHR	POR	SSNR					OCR	RDR	WDR	SSR	JOBA
3	PS	VS	STBS	POBS	STHS	POS	SSNS									
4																
5	DIE	UNQP	PUC	POC												
6																
7	BXLE	BAL	BTC	BXH	BFC	SIO	EL1	SET	RSET	RETN	CALL	ICOR	SRHL	SLHL	SRHA	RLH
8	BXLR	BALR	BTCR	BXHR	BFCR	SIOR										
9																
A																
B		PU	PUB	LSN	NH	CLH	OH	XH	LH	LB	AH	SH			ACH	SCH
C		PUR	PUBR	LSNR	NHR	CLHR	OHR	XHR	LHR	LBR	AHR	SHR			ACHR	SCHR
D		PUI	PUBI	LSNI	NHI	CLHI	OHI	XHI	LHI	LBI	AHI	SHI			ACHI	SCHI
E		PUS	PUBS	LSNS	NHS	CLHS	OHS	XHS	LHS	LBS	AHS	SHS			ACHS	SCHS
F		PUP	PUBP	LSNP	NHP	CLHP	OHP	XHP	LHP	LBP	AHP	SHP			ACHP	SCHP

APPENDIX II

INSTRUCTIONS LISTED ALPHABETICALLY BY OPCODE MNEMONIC

MNEMONIC	MEANING	NUMERIC	USE OF CONDITION CODE	PAGE
ACH	Add with carry halfword from core	BE	Use, Set	25
ACHI	Add with carry halfword immediate	DE	Use, Set	25
ACHP	Add with carry halfword from procedure	FE	Use, Set	25
ACHR	Add with carry halfword from register	CE	Use, Set	25
ACHS	Add with carry halfword from stream	EF	Use, Set	25
AH	Add halfword from core	BA	Use, Set	23
AHI	Add halfword immediate	DA	Use, Set	23
AHP	Add halfword from procedure	FA	Use, Set	23
AHR	Add halfword from register	CA	Use, Set	23
AHS	Add halfword from stream	EA	Use, Set	23
BAL	Branch and Link	71		48
BALR	Branch and Link from register	81		48
BFC	Branch on false condition	74	Use	47
BFCR	Branch on false condition	84	Use	47
BTC	Branch on true condition	72	Use	47
BTCR	Branch on true condition	82	Use	47

MNEMONIC	MEANING	NUMERIC	USE OF CONDITION CODE	PAGE
BXH	Branch on index high	73		48
BXHR	Branch on index high	83		48
BXLE	Branch on index low or equal	70		47
BXLR	Branch on index low or equal	80		49
CALL	Call procedure	7A		49
CLH	Compare logical halfword from core	B5	Set	30
CLHI	Compare logical halfword immediate	D5	Set	30
CLHP	Compare logical halfword from procedure	F5	Set	30
CLHR	Compare logical halfword from register	C5	Set	30
CLHS	Compare logical halfword from stream	E5	Set	30
DIE	Die	50		76
EL1	Enter level 1	76		75
ICOR	In core	7B	Set	77
JOBA	Job area location	2F		9
LB	Load byte from core	B9		21
LBI	Load byte immediate	D9		21
LBP	Load byte from procedure	F9		21
LBR	Load byte from register	C9		21
LBS	Load byte from stream	E9		21

MNEMONIC	MEANING	NUMERIC	USE OF CONDITION CODE	PAGE
LH	Load halfword from core	B8	Set	20
LHI	Load halfword immediate	D8	Set	20
LHP	Load halfword from procedure	F8	Set	20
LHR	Load halfword from register	C8	Set	20
LHS	Load halfword from stream	E8	Set	20
LSN	Load stream name from core	B3		18
LSNI	Load stream name immediate	D3		18
LSNP	Load stream name from procedure	F3		18
LSNR	Load stream name from register	C3		18
LSNS	Load stream name from stream	E3		18
NH	And halfword from core	B4	Set	27
NHI	And halfword immediate	D4	Set	27
NHP	And halfword from procedure	F4	Set	27
NHR	And halfword from register	C4	Set	27
NHS	And halfword from stream	E4	Set	27
OC	Output command from core	1B	Set	72
OCR	Output command from register	2B	Set	72
OH	Or halfword from core	B6	Set	28
OHI	Or halfword immediate	D6	Set	28

MNEMONIC	MEANING	NUMERIC	USE OF CONDITION CODE	PAGE
OHP	Or halfword from procedure	F6	Set	28
OHR	Or halfword from register	C6	Set	28
OHS	Or halfword from stream	E6	Set	28
P	P of semaphore in core	10		56
PO	Pop halfword into core	15		39
POB	Pop byte into core	13		39
POBR	Pop byte into register	23		40
POBS	Pop byte into stream	33		40
POC	Pop from control stack	53	Set	42
POR	Pop halfword into register	25		39
POS	Pop halfword into stream	35		39
PS	P of semaphore in stream	30		56
PU	Push halfword from core	B1		37
PUB	Push byte from core	B2		38
PUBI	Push byte immediate	D2		38
PUBP	Push byte from procedure	F2		38
PUBR	Push byte from register	C2		38
PUBS	Push byte from stream	E2		38
PUC	Push into control stack	52		42

MNEMONIC	MEANING	NUMERIC	USE OF CONDITION CODE	PAGE
PUI	Push halfword immediate	D1		37
PUP	Push halfword from procedure	F1		37
PUR	Push halfword from register	C1		37
PUS	Push halfword from stream	E1		37
RD	Read data into core	1C	Set	71
RDR	Read data into register	2C	Set	71
RETN	Return from procedure	79	Set	51
RLH	Rotate left halfword	7E	Set	31
RSET	Reset condition register	78		35
SCH	Subtract with carry halfword from core	BF	Use, Set	26
SCHI	Subtract with carry halfword immediate	DF	Use, Set	26
SCHP	Subtract with carry halfword from procedure	FF	Use, Set	26
SCHR	Subtract with carry halfword from register	CF	Use, Set	26
SCHS	Subtract with carry halfword from stream	EF	Use, Set	26
SET	Set condition register	77		35
SH	Subtract halfword from core	BB	Set	24
SHI	Subtract halfword immediate	DB	Set	24

MNEMONIC	MEANING	NUMERIC	USE OF CONDITION CODE	PAGE
SHP	Subtract halfword from procedure	FB	Set	24
SHR	Subtract halfword from register	CB	Set	24
SHS	Subtract halfword from stream	EB	Set	24
SIO	Start I/O channel	75		70
SIOR	Start I/O channel from register	85		70
SLHL	Shift left halfword logical	7D	Set	31
SRHA	Shift right halfword arithmetic	7E	Set	32
SRHL	Shift right halfword logical	7C	Set	32
SS	Sense status into core	1E	Set	72
SSN	Store stream name in core	16		19
SSNR	Store stream name in register	26		19
SSNS	Store stream name in stream	36		19
SSR	Sense status into register	2E	Set	73
STB	Store byte into core	12		22
STBR	Store byte into register	22		22
STBS	Store byte into stream	32		22
STH	Store halfword into core	14		22
STHR	Store halfword into register	24		22
STHS	Store halfword into stream	34		22

MNEMONIC	MEANING	NUMERIC	USE OF CONDITION CODE	PAGE
UNQP	Unqueue when level 1 finished	51		76
V	V of semaphore in core	11		56
VS	V of semaphore in stream	31		56
WD	Write data from core	1D	Set	71
WDR	Write data from register	2D	Set	71
XH	Exclusive or halfword from core	B7	Set	29
XHI	Exclusive or halfword immediate	D7	Set	29
XHP	Exclusive or halfword from procedure	F7	Set	29
XHR	Exclusive or halfword from register	C7	Set	29
XHS	Exclusive or halfword from stream	E7	Set	29

APPENDIX III

LOCATIONS IN THE JOB AREAS

Job Area Location	Contents	Name
0 - 1F	16 16-bit registers	RO - RF
20 - 3F	8 32-bit condition instructions	
40 - 4F	8 16-bit stream registers	SRO - SR7
50 - 51	Name of procedure stream being executed	JSTRNM
52 - 53	Name of control stack stream	CSNAME
54 - 55	Instruction counter (core value); only meaningful when job is not running (is waiting on a semaphore)	PC
57	Job number - condition code; only meaningful when job waiting on a semaphore	
5A - 5B	When an illegal instruction is encountered, TR5R6 may contain the value of the second operand	TR5R6
5C - 5D	Pointer to top of control stack (core value)	CSREG
5E	Stream page containing (first halfword of) instruction about to be executed	ICSTRP
60 - 63	Accumulated job run time	TIME1, TIME2
64 - 65	Input/Output semaphore	IOSEM
66 - 67	The condition register	COND

COND = | ON | MASK |
|---|---|
| 0 7 | 8 15 |

ON tells which conditions have occurred; MASK tells which are enabled. If $COND_{15}$ is on, the job is running in stream mode; otherwise it is running from core

Job Area Location	Contents	Name
68 - 69	The temporary instruction counter contains the core location of the instruction which would have been executed had an interrupt not occurred	TEMPIC
6A	LINK holds queue information if the job is not running	LINK
6B	PRIOR contains the priority of the job	PRIOR
6C - 6D	The microprogram saves μR4 and μR5 here while it is checking whether a page is in core	TR4R5
70 - 7B	The state of the micromachine is saved here when a program enters level 1. Of particular interest is the core location of the instruction after the one which caused entry into level 1; this is saved in 72 - 73	MICROSAVE
80 - 8F	8 16-bit extensions of the stream registers	SRX
90 - 91	Stream page/core page for alternate procedure page	JEXT
92 - 93	Stream page/core page for control stack	CSEXT
94 - 95	If bit 5 and bit 4 of BITS are on, then the RETN instruction is searching for an activation record. If bit 5 is on, but bit 4 is off, then level 1 is returning. If bit 6 is on, the job is participating in time-slicing	BITS
96	If $SAVREG_0 = 1$, then the control stack has overflowed/underflowed; otherwise $SAVREG_{3-5} =$ the number of the stream register being used to access the stack which overflowed/underflowed	SAVREG
99	If an attempt is made to execute an illegal condition instruction, its address relative to the job area is saved in OLDTP	OLDTP

Job Area Location	Contents	Name
9A	When an illegal instruction is encountered, TUTFF tells how long the instruction is; if it is 32 bits long, bit 6 will be on and bit 7 off; otherwise the instruction is 16 bits long and bit 7 is on and bit 6 off	TUTFF

APPENDIX IV

GLOBAL CORE LOCATIONS KNOWN TO THE MICROPROGRAM

Location	Contents	Name
9C - 9F	Time of last job change	CURRT1, CURRT2
A0 - A1	Semaphore controlling entry into level 1	DSKSEM
A2 - A3	Head of the queue of jobs which are ready to run	JOBSEM
A4 - A5	A4 contains the core page number, shifted right 1, of the job area of the most favored job; if there is no most favored job, $A4_0 = 1$. A5 contains the amount by which the priority of the most favored job is raised	MFJPR
A6	If bit 4 of A6 is on, the channel is enabled	MSW
A8 - A9	This halfword contains the address of the last halfword displayed	MAD
AA - AB	Core value of instruction counter before last power down	PROR1
AC	Number of running job before last power down and its condition code	PR2
AE - AF	Number of ticks per time-slice	TIMCNT
B0 - B1	Base for Device Status Table	NOVLIS
B2 - B3	Device number of clock used for time-slicing	CLKNUM
B4 - B7	Most recent time clock was read	CORET1, CORET2
100 - 1FF	Hash table	HASH
200 - 3FF	Age table	CPTAGE (INDCNT)
202 - 203	Head of the age chain	AGEHD

Location	Contents	Name
260 - 263	Accumulated idle time (implies core page 31 may not be used for page swapping)	IDLET1, IDLET2
400 - 5FF	Stream name table	SN
600 - 7FF	Stream page/link table	SPNEXT
109E - 10A1	First instruction of page fault	PGFLT
10A2 - 10A5	First instruction of level 1	LEVEL1

APPENDIX V

ILLEGAL OPCODES

This is the list of illegal opcodes for which the second operand has been developed and stored in TR5R6. The corresponding value of bit 6 of TUTFF is also given.

Opcode	Bit 6 of TUTFF
17, 18, 19, 1A	on
27, 28, 29, 2A	off
B0, BB, BC	on
C0, CB, CC	off
D0, DB, DC	on
E0, EB, EC	on
F0, FB, FC	on

APPENDIX VI

CONTROL STACK FORMATS

pointer before PUC →		
	data	
pointer after PUC →	unused	N*2
	0 7	8 15

PUC Record for PUC N

pointer before CALL →			
	(JSTRNM)		
	(SR1)		
	old instruction counter		
	(R1)		
	(R9)		
	unused	job number	condition code
pointer after CALL →	condition information	old condition MASK	
	0 7	8 15	

Activation Record for CALL

APPENDIX VII

CONTENTS OF CORE PAGE TABLE FOR CORE PAGES
WHICH CONTAIN STREAM PAGES

1. SN table. This table is located between 400 and 5FF. For each core page, there is one halfword of information:

0	1 15
	SN

Change Bit

Bit 0 of this halfword is set by the microprogram whenever data is stored in the core page. SN is the name of the stream whose page occupies this core page.

2. SPNEXT table. This table is located between 600 and 7FF. For each core page, there is one halfword of information:

0 7	8 15
SP	NEXT

SP is the number of the stream page which occupies this core page. NEXT is the core page of the next link on the hash chain. If NEXT = 0, this entry is the end of the chain.

3. INDCNT table. This table is located between 200 and 3FF. For each core page, there is one halfword of information:

0 7	8 15
IND	CNT

This information has two different meanings, depending on whether the stream page is locked in or not.

a. Locked in. The page is locked in if

IND = 0.

In this case CNT equals one less than the number of extensions locking in the page.

b. Aged. The page is on the age chain if

$$IND \neq 0.$$

In this case IND equals the core page of the next newer page on the age chain and CNT equals the number of the next older page.

APPENDIX VIII

INDEX TO LOCATIONS KNOWN TO THE MICROPROGRAM

Name	Type of Location	References
JOBSEM	GLOBAL	54, 55, 56, 76, 77, 78, 85
JSTRNM	JOB AREA	7, 20, 21, 23, 24, 25, 26, 27, 28, 29, 30, 37, 38, 44, 50, 52
LEVEL1	GLOBAL	74, 75
LINK	JOB AREA	9, 54, 55
MASK	JOB AREA	33, 50, 51, 75, 76
MFJPR	GLOBAL	78
MICROSAVE	JOB AREA	75, 76
MSW	GLOBAL	57, 85
NOVLIS	GLOBAL	59
ON	JOB AREA	33, 51
OLDTP	JOB AREA	35
PC	JOB AREA	44
PGFLT	GLOBAL	74, 75
PROR1	GLOBAL	83
PR2	GLOBAL	83
PRIOR	JOB AREA	54, 55, 75, 76
SAVREG	JOB AREA	36, 41
SN	GLOBAL	14, 16
SPNEXT	GLOBAL	15, 16
TEMPIC	JOB AREA	33
TIMCNT	GLOBAL	66, 69, 78

Name	Type of Location	References
TIME1	GLOBAL	78
TIME2	GLOBAL	78
TR4R5	JOB AREA	35, 75
TR5R6	JOB AREA	35
TUTFF	JOB AREA	35

Security Classification

DOCUMENT CONTROL DATA - R & D			
(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)			
1. ORIGINATING ACTIVITY *(Corporate author)* The MITRE Corporation Bedford, Massachusetts		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP	
3. REPORT TITLE PRINCIPLES OF OPERATION OF THE VENUS MICROPROGRAM			
4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)* N/A			
5. AUTHOR(S) *(First name, middle initial, last name)* Barbara J. Huberman			
6. REPORT DATE JULY 1970	7a. TOTAL NO. OF PAGES 112		7b. NO. OF REFS 0
8a. CONTRACT OR GRANT NO. F19(628)-68-C-0365 b. PROJECT NO. 700A c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-70-198 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* MTR-1843		
10. DISTRIBUTION STATEMENT This document has been approved for public release and sale; its distribution is unlimited.			
11. SUPPLEMENTARY NOTES N/A	12. SPONSORING MILITARY ACTIVITY Directorate of Planning and Technology, Electronic Systems Division, AF Systems Command, L. G. Hanscom Field, Bedford, Massachusetts		

13. ABSTRACT

Venus is a computer system comprised of microprograms and software. It is implemented on the Interdata 3, which is a small, microprogrammable computer. This document contains a complete description of the microprogram part of Venus.

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
VENUS MICROPROGRAMMING COMPUTER SOFTWARE VENUS MICROPROGRAM						

Security Classification